UNIVERSITÀ DEGLI STUDI DI PISA
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Anno Accademico 2002/2003

Tesi di Laurea

Simulazione di modelli di consistenza per file replicati su sistemi Grid

Candidato:
Gianni Pucciani

Relatori:
Ing. Andrea Domenici

Prof. Gianluca Dini

Indice

Capitolo 1

Introduzione

Lo scopo di questa tesi è quello di analizzare i problemi di consistenza dei file replicati su sistemi Grid e di progettare, simulare e mettere a confronto alcune possibili soluzioni.

Il sistema deve essere messo a disposizione del progetto European DataGrid, fondato dall'Unione Europea con l'obiettivo di creare una nuova infrastruttura di calcolo che dia alle comunità scientifiche di tutto il mondo la possibilità di analizzare grandi quantità di dati largamente distribuiti. Il progetto è guidato dal CERN (European Organization for Nuclear Research) di Ginevra, assieme ad altri partner provenienti da tutta Europa.

Una delle caratteristiche dei sistemi Grid è quella di consentire l'accesso, semplice, sicuro e coordinato, ad una quantità di dati enorme (dell'ordine dei PetaByte) distribuiti nei vari nodi del sistema e di mettere a disposizione un'adeguata potenza di calcolo per effettuare le necessarie elaborazioni su di essi. Per migliorare l'accesso ai dati si ricorre a tecniche di replicazione che, se da un lato introducono grossi vantaggi, dall'altro incrementano la mole di dati da gestire ed introducono nuove problematiche di gestione, come quella della consistenza. L'ipotesi di considerare questi dati di tipo read-only, accettata nelle fasi iniziali del progetto, si è rivelata troppo grossolana, poiché esiste una parte di dati (e metadati) che devono necessariamente consentire ad un utente o al sistema stesso di poter aggiungere e/o modificare alcune informazioni. La possibilità di aggiornare una replica di un certo file fa sì che le altre repliche possano venire a trovarsi in uno stato inconsistente, per cui un altro utente del sistema potrebbe accedere ad informazioni non aggiornate.

Questo è il problema della consistenza che ci apprestiamo ad analizzare. In questa sede non siamo interessati a proporre una soluzione ad alte prestazioni, bensì a studiare il problema e a proporre alcune soluzioni, che saranno valutate tramite simulazioni.

1.1 Struttura dei capitoli

Diamo uno sguardo all'organizzazione dei vari capitoli.

Nel Capitolo 2 introdurremo il concetto di Grid System. Vedremo cosa sono i sistemi Grid, come è iniziata la ricerca in questo settore e quali sono le problematiche che intende affrontare. Il settore della Grid research è in rapida evoluzione; progetti ed iniziative di vario tipo nascono frequentemente in ogni parte del mondo. Cercheremo di dare delle informazioni di base sui progetti di maggior interesse e visibilità.

Nel Capitolo 3 vedremo più da vicino il progetto European DataGrid (EDG), com'è strutturato, che tipo di prodotto cerca di offrire e a chi in particolare. Questo ci permetterà di introdurre alcuni concetti importanti per il Grid Computing e di vedere come realmente un utente può utilizzare una Griglia basata su EDG.

Con il Capitolo 4 ci addentreremo nelle problematiche del data management, in particolare studieremo il meccanismo della replicazione dei dati e i possibili modi di risolvere il problema della consistenza. La replicazione è una tecnica ben nota nel campo dei sistemi distribuiti, in particolare nei database e nei filesystem distribuiti. Vedremo quali tecniche vengono utilizzate in questi settori per risolvere il problema della consistenza e cercheremo di capire se queste tecniche possono essere utilizzate per risolvere lo stesso problema in una Griglia.

Il Capitolo 5 presenta un approfondimento delle questioni relative al data management all'interno di EDG. Vedremo i vari progetti e le soluzioni vecchie e nuove adottate all'interno del WP2, il gruppo di lavoro di EDG che si occupa del data management. Questo ci permetterà di capire le basi su cui costruiremo i nostri modelli, che sono argomento del prossimo capitolo.

Nel Capitolo 6 vedremo come è possibile simulare il funzionamento di una Griglia. In particolare vedremo il funzionamento del simulatore OptorSim ed il suo originale utilizzo come simulatore di algoritmi di ottimizzazione.

Nel Capitolo 7 vedremo come è possibile utilizzare OptorSim per simulare un Servizio di Consistenza delle repliche. Vedremo alcuni dettagli implementativi sui modelli che abbiamo scelto di simulare ed il loro utilizzo.

Il Capitolo 8 riassume il lavoro svolto e quello futuro che potrà seguire a questa tesi.

Capitolo 2

Griglie: cosa sono e a cosa servono

In questo capitolo vedremo che cos'è una Griglia, come è nato il concetto di Grid system e con quali propositi, evidenziando le differenze con le altre applicazioni distribuite esistenti. Introdurremo il concetto di Virtual Organization, un sistema di collaborazioni destinato a cambiare il modo di pensare la ricerca scientifica, dopo di che analizzeremo la struttura interna di una griglia, per capire come i vari servizi e protocolli interagiscono. Cercheremo poi di dare uno sguardo ai principali progetti in corso che riguardano le Griglie evidenziando le iniziative più importanti, principalmente Globus, lo standard di fatto per quanto riguarda il middleware e il Global Grid Forum, che cerca di portare avanti in maniera coordinata la ricerca in questo nuovo settore.

2.1 Cenni storici

In [25], che può considerarsi il manifesto della ricerca Grid, si parla di Grid come una nuova infrastruttura di calcolo che cambierà il nostro modo di pensare ed usare il computer. In questo caso, con il termine "The Grid", si indica una Griglia globale, che dovrebbe collegare numerose Griglie computazionali di livello locale, nazionali o regionali, al fine di creare un'unica sorgente universale di potenza di calcolo. È in questo senso che si può ritrovare il significato del termine Grid, scelto per analogia con la rete di potenza elettrica, che fornisce accesso all'energia elettrica. Si pensa infatti che, come la diffusione dell'energia elettrica ha avuto un grosso impatto sullo sviluppo industriale e sociale della società, in maniera analoga un accesso semplice, capillare ed economico a potenza di calcolo, di memoria, informazioni e quant'altro riconducibile all'utilizzo dei computers, dovrebbe favorire lo sviluppo e la nascita di nuove classi di applicazioni informatiche. La rete elettrica non è l'unico esempio a cui possiamo paragonare le griglie; altre infrastrutture simili sono quella autostradale e quella telefonica. Se diamo uno sguardo al passato,

cercando di valutare le condizioni che precedevano e che sono seguite allo sviluppo di queste infrastrutture, è quanto meno affascinante pensare alle innovazioni scientifiche e sociali che possono seguire allo sviluppo di una Griglia computazionale. Sebbene l'uso dell'energia elettrica, del telefono e delle reti stradali siano ormai alla portata di tutti (o almeno dovrebbero esserlo ...), è prevedibile che l'utilizzo delle griglie, almeno inizialmente, sia rivolto per lo più alle comunità scientifiche, anche perché, come già accennato, è difficile, ad oggi, prevedere quali applicazioni potranno sorgere un domani per giustificare l'utilizzo di massa di una Griglia.

Il termine "Grid" nasce quindi intorno alla metà degli anni novanta. In quegli anni si riteneva infatti che lo stato dell'arte nei vari settori dell'informatica e delle telecomunicazioni fosse ormai pronto per concretizzare l'idea delle Griglie. Si disponeva infatti di reti ad alta velocità, potenti microprocessori ed architetture parallele, protocolli di comunicazione ben consolidati, meccanismi di sicurezza, tecniche di commercio elettronico ed applicazioni di vario genere. Tutto ciò doveva costituire la base su cui costruire una Griglia. La scommessa era quindi quella di riunire le varie esperienze di settore e farle collaborare per dare vita a qualcosa di nuovo.

2.2 Organizzazioni Virtuali

In [27] si definisce il "Grid problem" come ricerca di un modo flessibile, sicuro e coordinato per condividere risorse tra gruppi dinamici di individui, istituzioni e risorse, ovvero tra Organizzazioni Virtuali (Virtual Organization, VO).

Il concetto di "Virtual Organization" è stato introdotto contemporaneamente a quello di "Grid Computing", e serve a sottolineare la nuova natura dei legami che uniscono varie persone che partecipano allo sviluppo di uno o più progetti tramite l'utilizzo di una Griglia. Due VO possono differire per quanto riguarda il numero e il tipo dei partecipanti, la durata e la scala delle loro interazioni, le risorse condivise e le modalità di condivisione. Infatti, ogni membro di una VO, che può essere un laboratorio di ricerca, una università, una ditta privata etc., nel mettere a disposizione le proprie risorse, hardware, software ma anche forza lavoro, può porre dei vincoli sul loro utilizzo e su cosa può essere fatto, quando, come e da chi. L'implementazione di questi vincoli richiede dei meccanismi per esprimere le politiche di utilizzo di una particolare risorsa, per stabilire le identità di chi vi accede e per determinare se una data operazione infrange le regole stabilite. I vari partecipanti ad una VO devono quindi convenire riguardo l'uso di particolari politiche incluse quelle relative alla sicurezza. Ogni persona partecipante al progetto può così sviluppare il proprio lavoro indipendentemente dalla sua posizione geografica e dagli strumenti hardware/software usati, avendo a disposizione tutti i dati e la potenza di calcolo di cui necessita e potendovi accedere in maniera semplice e trasparente. Come detto, la natura dei legami che uniscono i vari membri di una VO ed i vincoli imposti sull'utilizzo delle

risorse comuni sono dinamici; possono variare nel tempo non solo i partecipanti ad una VO ma anche le risorse a disposizione e le loro modalità di utilizzo. Può variare nel tempo anche il ruolo di un partecipante: produttori possono diventare consumatori e viceversa.

Le architetture Grid si differenziano dai sistemi di condivisione di risorse e di calcolo distribuito già presenti ed operanti nel settore dell'IT. Una prima differenza risiede nelle risorse da condividere: in molti casi ci si limita allo scambio di dati o all'utilizzo di particolari programmi applicativi, spesso facendo leva sull'uso di server centralizzati. Appartengono a questa categoria le applicazioni di B2B Exchange. Un'altra distinzione può essere fatta circa l'ambiente di condivisione, non più ristretto a singole aziende che adottano una piattaforma comune come J2EE o DCOM, ma più esteso ed eterogeneo. Anche la modalità di accesso alle risorse è differente, non più centralizzato ed in maniera più o meno privata (ASP [1], SSP [2]). In sostanza le tecnologie attuali, in un modo o nell'altro, non rispondono alle esigenze di flessibilità e controllo sulle risorse condivise proprie delle Organizzazioni Virtuali. Tuttavia è bene precisare che le tecnologie Grid non si propongono come alternativa alle soluzioni di calcolo distribuito esistenti, ma cercano di complementarne l'utilizzo.

2.3 Architettura di una Griglia

Una Griglia può essere vista come un insieme di protocolli, servizi ed API (Application Programming Interface) [27].

Date le differenze che contraddistinguono i membri di una VO, uno dei problemi principali da affrontare nella costruzione di una Griglia è l'interoperabilità, che in un ambiente di rete significa utilizzare protocolli comuni. Una Griglia quindi è soprattutto una architettura di protocolli, che permettono agli utenti di negoziare, stabilire e controllare le risorse condivise. Così come il Web ha rivoluzionato il modo di scambiare le informazioni tramite l'uso di protocolli e formati standard (HTTP e HTML principalmente), allo stesso modo una Griglia necessita di protocolli per la condivisione di risorse. Dato che le VO devono complementare piuttosto che sostituire le soluzioni informatiche esistenti, i nuovi meccanismi di condivisione non possono imporre un radicale cambiamento delle politiche locali e devono permettere alle varie istituzioni di mantenere il controllo sulle proprie risorse.

Una Griglia è anche un insieme di servizi standard, utilizzati per accedere alle risorse di calcolo ed eseguire delle operazioni, per reperire informazioni, schedulare processi, replicare i dati e così via. L'utilizzo di questi servizi ha lo scopo di nascondere all'applicazione

[1] Application Service Provider
[2] Storage Service Provider

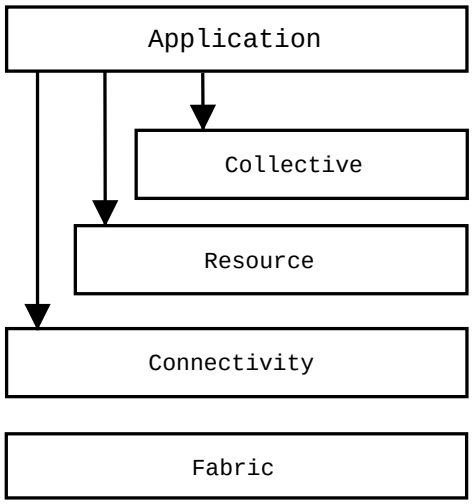


Figura 2.1: Architettura a livelli dei protocolli Grid

utente finale le operazioni necessarie a svolgere una determinata operazione. Il progetto che svilupperemo nel corso di questa tesi ha proprio l'obiettivo di creare un servizio, quello della consistenza dei dati replicati.

Infine una Griglia deve anche fornire il materiale necessario per costruire applicazioni che devono girare al suo interno. Il materiale in questione sono API e SDK (Software Development Kit). Questi strumenti devono favorire ed accelerare lo sviluppo di applicazioni Grid-aware [3].

Questi tre componenti basilari di una Griglia, protocolli, servizi e API sono altamente correlati. Ognuno ha bisogno dell'altro per svolgere il proprio lavoro.

2.3.1 Visione stratificata dell'architettura Grid

L'architettura dei protocolli di una Griglia può essere visualizzata, così come accade per i modelli OSI o TCP/IP, in una struttura a livelli come quella mostrata in figura 2.1.

Vediamo il significato di questi livelli. Il Fabric layer fornisce le risorse il cui accesso è mediato da parte dei protocolli sovrastanti. CPU, storage systems, cataloghi, risorse di rete e sensori sono alcuni esempi di queste risorse. Una risorsa non è necessariamente un componente hardware: anche una entità logica come un filesystem distribuito può appartenere al Fabric layer. È importante non imporre dei requisiti troppo stringenti sulle funzionalità che devono implementare queste risorse per facilitare l'integrazione con le tecnologie già esistenti.

[3] Applicazioni che vengono eseguite con l'ausilio di una Griglia

Il Connectivity layer fornisce i protocolli di base per la comunicazione semplice e sicura tra le risorse del Fabric layer. Questo livello sfrutta i protocolli già consolidati come quelli del modello TCP/IP, così come per la parte che riguarda la sicurezza si cerca sempre di sfruttare gli standard esistenti per l'autenticazione, la protezione delle comunicazioni e l'autorizzazione.

Il Resource layer fa leva sui protocolli del livello sottostante per implementare l'accesso e il controllo delle risorse locali, ignorando le problematiche relative allo stato globale del sistema.

A differenza del Resource layer, che è concentrato sull'utilizzo di ciascuna singola risorsa, il Collective layer contiene protocolli e servizi che riguardano il sistema nella sua globalità e le interazioni tra i vari componenti. A questo livello troviamo servizi che non solo permettono di trovare le risorse (directory services) in base a particolari attributi come la tipologia, la disponibilità e il carico ma anche di effettuare operazioni di allocazione, scheduling e brokering, servizi di monitoraggio e diagnostica per controllare lo stato delle risorse. È in questo strato che si colloca anche il Data Replication Service all'interno del quale si articola il lavoro di questa tesi.

L'ultimo livello dell'architettura è costituito dall'Application layer, che comprende le applicazioni utente che operano nel contesto di una VO.

I tre strati centrali, Connectivity, Resource e Collective, formano quello che generalmente viene chiamato il "middleware"[4], ovvero la colla che unisce il tutto. Il progetto Globus, con il suo Toolkit, che vedremo meglio nella prossima sezione, rappresenta la principale iniziativa a livello mondiale per lo sviluppo di un Grid middleware.

2.4 Progetti e iniziative Grid nel mondo

Come già accennato il campo della ricerca Grid è in rapida evoluzione ed iniziative di vario genere stanno prendendo piede un po' in tutto il mondo. Uno dei progetti più importanti è quello che fa capo al gruppo Globus [6], che con il suo Toolkit si appresta a diventare lo standard per quanto riguarda i servizi di middleware. I partners del progetto Globus sono: l'Argonne National Laboratory, l'Istituto di Informatica dell'Università di Southern California, l'Università di Chicago, l'Università di Edimburgo e lo Swedish Center for Parallel Computers.

Il Globus Toolkit raccoglie servizi e librerie che riguardano il resource monitoring, discovery e management, la sicurezza e la gestione dei file. La sua ultima versione, il Globus Toolkit 3 (GT3), è una implementazione interamente basata sul modello OGSA (Open

[4]Con questo termine si indica quell'insieme di servizi necessari a supportare un set di applicazioni in un ambiente di rete distribuito

Grid Service Architecture [1]. Il Toolkit è fatto di componenti che possono essere usati singolarmente o tutti insieme per creare applicazioni Grid. La politica open-source[5] con cui è portato avanti il progetto ne facilita l'utilizzo e contemporaneamente contribuisce in maniera significativa alla continua crescita del prodotto. La versione 1.0 risale al 1998 mentre la versione 3.0 è stata rilasciata nell'Agosto del 2003. Il GT è utilizzato nei principali progetti Grid mondiali: European DataGrid, di cui parleremo approfonditamente nei prossimi capitoli, NASA's Information Power Grid [11], progetti Grid della NSF come NPACI [13], Grid Physics Network (GriPhyN [8]) e Particle Physics Data Grid [14]. Altri importanti progetti che utilizzano Globus sono il Network for Eartquake Engineering and Simulation (NEES [12]), FusionGrid [3] e Earth System Grid [2]. Da qualche anno progetti Grid basati su Globus sono portati avanti nei settori commerciali da Avaki, HP, IBM, NEC, Oracle, Sun ed altri. L'utilizzo di Globus nell'industria ha portato una nuova serie di obiettivi, che tuttavia non devono entrare in conflitto con l'originale strategia open source del progetto.

Ogni componente del GT viene fornito con delle API in linguaggio C per essere usato dagli sviluppatori. Molti di essi sono accompagnati da strumenti a linea di comando e, per i più importanti da API Java.

Parlando delle principali iniziative relative alla ricerca Grid nel mondo non si può non accennare al Global Grid Forum (GGF [4]), il forum mondiale che raccoglie le varie esperienze della ricerca Grid. Il suo scopo è quello di promuovere e supportare lo sviluppo e l'implementazione di tecnologie ed applicazioni Grid attraverso la raccolta di documenti di *best practice* e specifiche tecniche e tramite la pubblicazione e il confronto diretto delle varie proposte. Partecipano al GGF più di 5000 persone, tra ricercatori e lavoratori del settore, provenienti da oltre 50 paesi. I meeting del Forum, che si svolgono più volte l'anno in diversi paesi, e a cui partecipano confrontandosi centinaia di persone, sono un'ottima occasione per creare dei legami tra le varie esperienze locali.

[5] L'open-source è la strategia di sviluppo software propria dei sistemi Unix/Linux. L'idea di base è quella di migliorare la qualità di un prodotto software attraverso la libertà, da parte dei programmatori, di leggere, modificare e ridistribuire il codice sorgente

Capitolo 3

Il progetto European DataGrid

In questo capitolo presenteremo il progetto European DataGrid, cercheremo di capire quali sono i suoi obiettivi, come è organizzato e vedremo alcuni dettagli del prodotto che offre. Questo ci permetterà di capire meglio alcuni aspetti del Grid Computing e di chiarire il punto di vista dell'applicazione utente rispetto a un tale sistema.

3.1 Presentazione del progetto

Come già detto il progetto EU DataGrid è guidato dal CERN, ma tra i partners principali figurano anche l'ESA (Agenzia Spaziale Europea), il CNRS (le Centre National de la Richerche Scientifique, FR), l'INFN (Istituto Nazionale di Fisica Nucleare, IT), il NIKHEF (the national institute for subatomic physics in the Netherlands) e il PPARC (Particle Physics and Astronomy Research Council, UK). Altri quindici partners provenienti da tutta Europa partecipano come associati.

Le nuove frontiere della ricerca scientifica richiedono una potenza di calcolo e di immagazzinamento dati che non possono essere fornite da una singola istituzione. Il progetto ha lo scopo di mettere in atto una soluzione che consenta alle varie organizzazioni di collaborare in un campo specifico, unendo le loro capacità in termini di risorse. Questa nuova infrastruttura sarà messa a disposizione per analizzare dati provenienti da esperimenti scientifici nei campi della fisica delle alte energie, la biologia e il monitoraggio terrestre. Devono inoltre essere sviluppare una serie di iniziative atte a favorire e coordinare le collaborazioni tra organizzazioni e ricercatori, indipendentemente dalla loro dislocazione geografica.

DataGrid è un progetto ambizioso, iniziato nel 2001 per una durata di 3 anni, con più di 200 scienziati e ricercatori coinvolti e con una disponibilità di fondi da parte dell'Unione Europea che sfiora i dieci milioni di euro.

3.1.1 Gruppi di lavoro di EDG

Il progetto è stato diviso in 12 Work Package (WP), distribuiti su quattro Working Group (WG): Middleware, Infrastructure, Applications e Management. L'organizzazione dei vari WP con il relativo WG è riassunto nel seguente elenco:

- Middleware

  WP1 Grid Work Scheduling: ha lo scopo di definire ed implementare un architettura per lo scheduling ed il resource management per raccogliere ed organizzare il carico di lavoro, caotico ed imprevedibile, generato dagli utenti della griglia. Un esempio di task svolto da questo WP è quello di individuare e schedulare una operazione in un certo nodo del sistema, possibilmente quello migliore, in relazione alla disponibilità di dati ed alle condizioni di traffico in rete.

  WP2 Data Management: è il gruppo di lavoro all'interno del quale si articola il lavoro di questa tesi. Lo scopo di questo WP, i cui lavori vedremo in dettaglio nei prossimi capitoli, è quello di produrre dei servizi di middleware per gestire e far condividere il grosso volume di dati prodotto dalle varie applicazioni. Tratta argomenti come la sicurezza dei dati, il loro trasferimento fra diversi siti, la replicazione e la consistenza.

  WP3 Grid Monitoring Services: si occupa dello sviluppo di servizi per permettere ad utenti ed amministratori di accedere informazioni di stato o di errore, per monitorare costantemente il funzionamento della Griglia, facilitare le fasi di ottimizzazione dei job e consentire l'individuazione e la risoluzione dei problemi.

  WP4 Fabric Management: questo WP è incaricato della gestione delle risorse del fabric layer (vedi sezione 2.3.1) di tipo computazionale e lo sviluppo di strumenti per facilitare l'impiego e la gestione di potenti sistemi di calcolo.

  WP5 Mass Storage Management: come il precedente WP, questo affronta le problematiche di gestione delle risorse del fabric layer. Invece di trattare l'hardware computazionale, il WP5 si occupa degli Storage Systems, tutti quei sistemi atti ad immagazzinare dati. Data la notevole mole di dati in gioco si capisce che le usuali tecniche di immagazzinamento dati (DDBMS, MSS) non sono sufficienti a soddisfare le nuove esigenze oppure necessitano di un grande lavoro di adattamento.

- Infrastructure

  WP6 Integration Testbed & Support: il WP6 si occupa dell'organizzazione dei testbed e dello sviluppo della documentazione necessaria per l'uso della Griglia da parte degli utenti finali.

WP7 Network Services: come suggerisce il nome si occupa dell'implementazione dei servizi di rete ma tratta anche aspetti legati alla sicurezza e al monitoraggio del traffico di rete, e quelli relativi alla pianificazione dei testbed in collaborazione con il WP6.

- Applications

  WP8 HEP

  WP9 Earth Observation

  WP10 Biology

  Sono i tre gruppi di lavoro che devono fungere da interfaccia tra gli sviluppatori dei servizi e gli scienziati di queste tre aree di ricerca che devono sfruttare la Griglia per svolgere le loro ricerche. Questo lavoro assume una notevole rilevanza nella fase di raccolta dei requisiti delle applicazioni utente.

- Management

  WP11 Dissemination

  WP12 Project Management

  Questi due ultimi work package sono quelli che si occupano di gestire, coordinare e pubblicare il lavoro svolto dai singoli work package.

3.2 Testbed

All'interno del progetto svolgono un ruolo fondamentale le operazioni che riguardano i testbed. Le fasi di Testbed sono quelle in cui vengono testati i servizi implementati all'interno del progetto. In pratica il Testbed è la realizzazione di una Griglia di prova in cui vengono svolte una serie di operazioni per valutare la correttezza e la bontà dei servizi e dei protocolli implementati.

Nella figura 3.1 possiamo vedere le risorse coinvolte in un semplice testbed con 3 siti ed una Certfication Authority (CA). La CA si incarica di rilasciare i certificati [1] per qualsiasi utente che vuole svolgere delle attività tramite la Griglia. Non solo gli utenti ma anche le risorse hardware ed ogni attore coinvolto nelle operazioni della Griglia deve ottenere un certificato per provare la propria identità. Un Testbed è costituito quindi da più siti. Ogni sito contiene un certo numero di macchine, ed ogni macchina svolge il proprio ruolo, a

[1] Il certificato è un file contenente alcuni dati identificativi di una persona, in un contesto determinato, abbinati alla chiave pubblica della stessa, firmato da una o più autorità di certificazione. In pratica le firme di queste autorità servono a garantire la veridicità dei dati

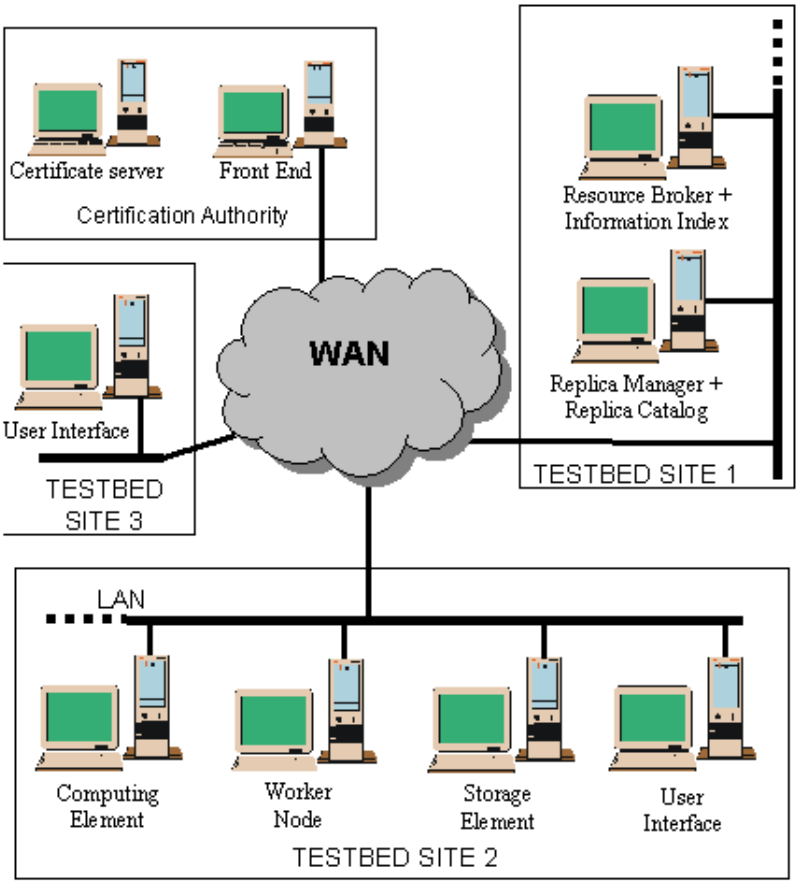


Figura 3.1: Architettura di testbed con 3 siti e una CA

seconda del modulo di middleware installato su quella macchina. Vediamo una lista dei possibili ruoli:

- Resource Broker(RB): è il modulo che riceve le richieste degli utenti ed effettua delle query verso l'Information Index per trovare le risorse più adatte a svolgere l'operazione richiesta dall'applicazione utente.
- Main Discovery Service: agisce come punto di raccolta di tutte le informazioni presenti e visibili nel testbed. L'Information Index (II) invece funziona da cache per le informazioni residenti nel MDS, e riceve le richieste dal RB e dagli utenti finali tramite le User Interfaces.
- Replica Manager(RM): lo vedremo in dettaglio nei prossimi capitoli; per adesso è sufficiente sapere che si occupa della replicazione dei dati nei vari siti per muovere le informazioni più vicino possibile al luogo dove verranno utilizzate.
- Computing Element(CE): è il modulo che riceve le richieste di esecuzione dei job e le inoltra ai vari worker nodes (WN) dove avverrano le necessarie elaborazioni. Un CE in realtà è costituito da un Gate Keeper (GK), che funge da interfaccia verso le risorse computazionali vere e proprie, rappresentate da uno o più WN che possono essere installati sulla stessa macchina del CE.
- Storage Element(SE): è il modulo installato sulle macchine che ospiteranno i dati. Deve fornire una interfaccia uniforme verso differenti tipi di storage system.
- User Interface(UI): è il modulo che permette agli utenti di utilizzare i servizi implementati nella Griglia: job submission, data management, information management, etc.

3.2.1 Guida utente

In questa sezione vediamo cosa deve fare un utente per utilizzare le risorse impegnate nel testbed. Questa parte ci aiuterà a comprendere meglio come sarà effettivamente usata una vera Griglia. Per informazioni dettagliate fare riferimento alla guida utente di EDG [22].

3.2.1.1 Registrazione e creazione del proxy

La prima cosa che un utente deve fare per poter sfruttare le risorse della Griglia[2] è quella di ottenere un certificato da una Certification Authority approvata dal progetto EDG[3]. Il

[2]Nel seguito parleremo indifferentemente di Griglia o testbed

[3]per una futura Griglia la CA dovrà essere approvata dall'organo di riferimento che amministra la Griglia

certificato è l'equivalente Grid di un passaporto e serve per attestare l'identità di un utente o di una risorsa coinvolta nel testbed.

Con il certificato installato nel browser l'applicazione utente deve registrarsi presso una Virtual Organization. Da questo momento in poi, ogni lavoro eseguito nella Griglia per conto di questo utente, sarà eseguito in un ambiente specifico legato alla VO di appartenenza.

Per immettere dei job nella griglia l'applicazione utente deve ottenere un account su una macchina in cui è installato il software necessario per entrare nella Griglia, ovvero il modulo UI. La macchina può essere locale all'istituzione di appartenenza dell'utente (ad esempio una Università) oppure remota, ospitata al CERN.

Il certificato serve anche a stabilire i diritti di accesso di un utente verso le risorse della Griglia. L'accesso è controllato tramite un proxy che contiene delle credenziali firmate dall'applicazione utente e valide per un limitato periodo di tempo[4]. I servizi Grid possono svolgere operazioni per conto dell'utente solo se hanno una copia di questo proxy. Un proxy è generato automaticamente chiamando un comando da console (*grid-proxy-init*) con il certificato correttamente installato sulla macchina. Altri comandi ci permettono di ottenere informazioni sul proxy, invalidarlo, etc.

3.2.1.2 Job submission

Il sistema di job submission fornisce i comandi necessari per effettuare il submit dei job, controllarli, gestirli e ricevere l'output da loro generato. Il sistema permette all'applicazione utente di fare il submit di un job dopodiché si incarica di trovare, all'interno del testbed, le risorse più appropriate per eseguirlo. Con la figura 3.2 possiamo fissare meglio le idee per quanto riguarda il sistema di job submission, i vari componenti coinvolti e le loro interazioni.

Alcuni dei comandi utilizzati per gestire i job sono:
edg-job-submit <job.jdl>
edg-job-status <jobId>
edg-job-get-output <jobId>
edg-job-cancel <jobId>
Il primo comando serve per fare il submit di un job tramite una sua descrizione contenuta in un file con estensione .jdl. Questo file è scritto con un linguaggio ad hoc, il *Job Description Language (JDL)* [36], e contiene tutte le informazioni necessarie all'esecuzione del job: dati di ingresso, di uscita, requisiti di risorse, etc.

[4] 12 ore è il valore di dafault

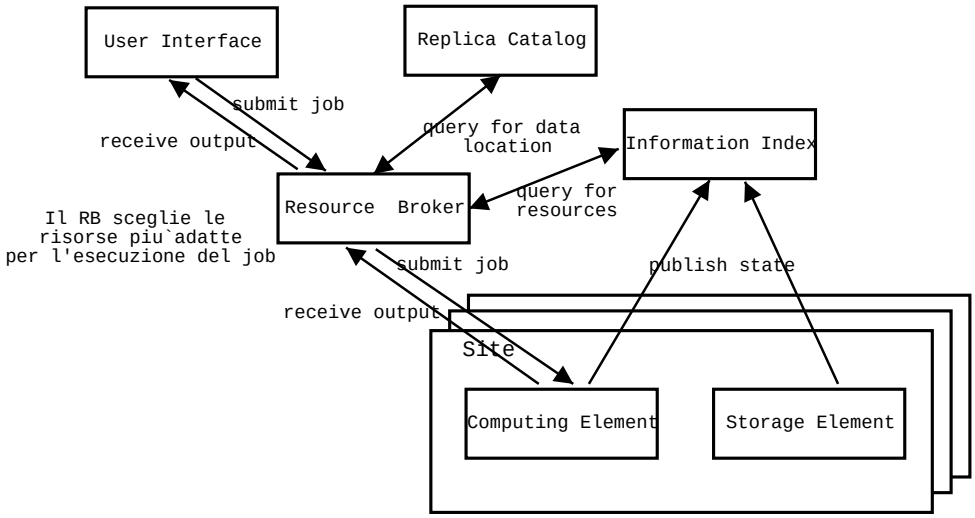


Figura 3.2: Sistema di job submission

3.2.1.3 Trasferimento e gestione dei file

Il trasferimento manuale di file tra diversi nodi del sistema può essere fatto tramite il comando:
globus-url-copy [options] sourceURL destURL
che permette anche il cosiddetto *third-party transfer*, ovvero il trasferimento di un file tra due nodi del sistema che sono remoti rispetto a chi esegue il comando.

Per la gestione di file e directory ci sono dei comandi analoghi a quelli dei sistemi unix:
edg-gridftp-exists
edg-gridftp-ls
edg-gridftp-mkdir
edg-gridftp-rename
edg-gridftp-rm
edg-gridftp-rmdir

3.2.1.4 Replicazione dei file e catalogo delle repliche

Come vedremo in dettaglio nei prossimi due capitoli, un buon sistema di replicazione e di gestione delle repliche è fondamentale per migliorare le prestazioni di un sistema distribuito. Solo i file presenti in uno Storage Element possono essere registrati nel catalogo (RC) entrando così a far parte della Griglia. Per inserire un file nella Griglia un utente deve prima di tutto trovare uno SE su cui egli ha diritto di scrittura. Queste informazioni sono reperibili dall' Information Index o direttamente dal Replica Catalog. Una volta

conosciuto lo SE su cui vogliamo trasferire i file (host e path), tramite il software EDG, e precisamente con i comandi:
globus-url-copy oppure *edg-replica-manager.copyFile*
un file presente nella macchina dell'applicazione utente può essere trasferito nella Griglia. Per renderlo effettivamente visibile ed utilizzabile dagli altri membri della sua VO, è necessario un altro passo: il file deve essere registrato nel catalogo delle repliche tramite il comando:
edg-replica-manager.registerEntry
È possibile inoltre effettuare delle query direttamente al RC per ricevere informazioni sulla posizione di alcuni file o sui loro attributi, copiare file da un CE (WN) ad un SE e replicare file da un SE ad un altro. Si rimanda come al solito alla EDG User's Guide [22] per un elenco completo.

3.3 Applicazioni

Vediamo adesso in che modo una Griglia può essere sfruttata dalla ricerca scientifica. Fisica delle alte energie (HEP), biologia e monitoraggio terrestre sono le tre aree di ricerca che collaborano attivamente allo sviluppo del software EDG. Vediamo quali sono le peculiarità di queste discipline e quali le richieste che impongono agli sviluppatori dei servizi di middleware.

3.3.1 DataGrid e l'High Energy Physics

Uno dei campi di maggior interesse nella fisica delle alte energie è quello che riguarda le particelle fondamentali della materia e le forze che agiscono tra di loro. In particolare si cerca di capire perché alcune particelle sono "più forti" di altre e se la risposta ad alcuni di questi quesiti può risiedere nella presenza, ancora non provata, dei cosiddetti "Higgs field". Per questo ed altri scopi, a Ginevra, presso il CERN, è situato l'LEP (Large Electron Positron accelerator), un acceleratore di particelle (il più grande del mondo) che, mediante collisioni, produce una grande mole di dati utilizzati da molti esperimenti nel campo della ricerca nucleare.

In figura 3.3 e 3.4 vediamo la posizione geografica del tunnel che ospita l'acceleratore e l'indicazione dei vari siti che ospitano i rilevatori. In figura 3.5 è mostrata una sezione del rilevatore di particelle utilizzato nell'esperimento Alice. Nel 2006, nello stesso tunnel che ospita l'LEP (a 100m di profondità per un diametro di circa 26Km), entrerà in funzione un nuovo acceleratore, l'LHC (Large Hadron Collider) e con esso partiranno quattro nuovi esperimenti di HEP (High Energy Physics): ALICE, ATLAS, CMS e LHCb, ognuno dei quali produrrà una grande mole di dati, dell'ordine dei Petabytes all'anno, dalla cui analisi la comunità scientifica si aspetta di compiere notevoli passi avanti nella ricerche di HEP.

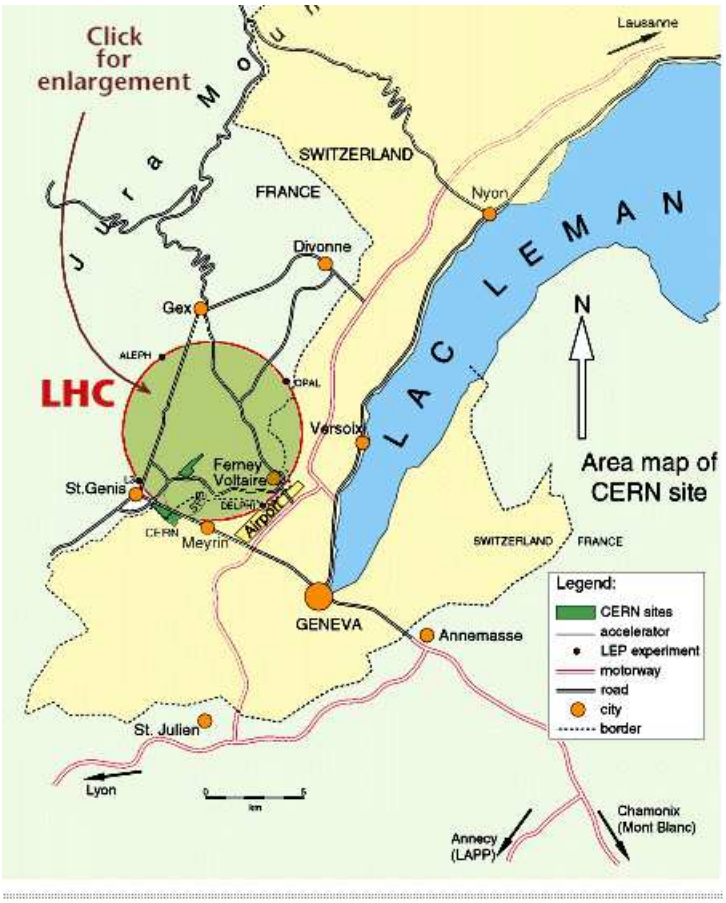


Figura 3.3: Collocazione geografica del Tunnel che ospita l'LEP

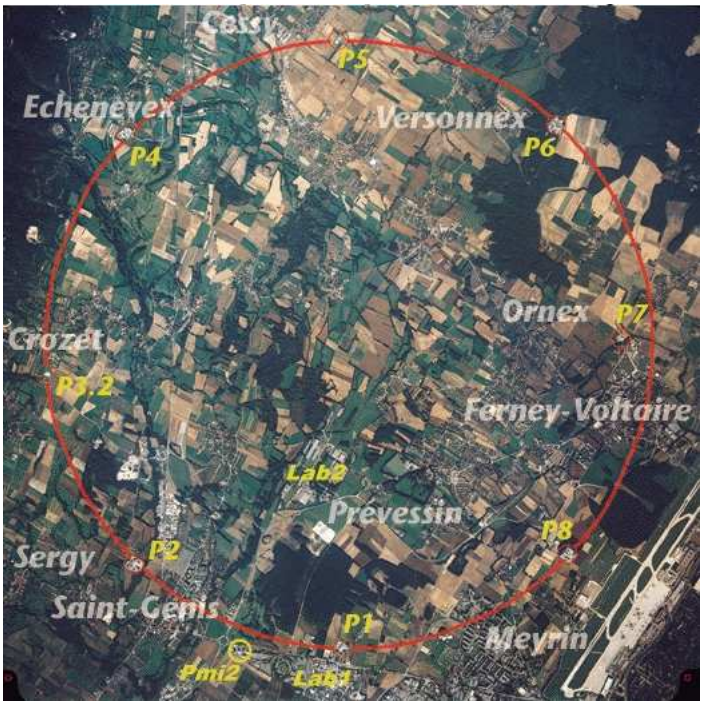


Figura 3.4: Visione aerea del tunnel

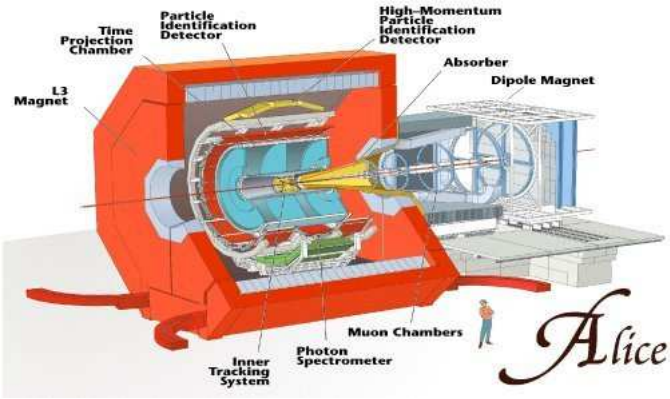


Figura 3.5: Il Rilevatore di particelle utilizzato in Alice

Tabella 3.1: Caratterizzazione dei dati in un esperimento di HEP

data type	size	storage amount per year
raw data	1MB	1PB
storage amount per year	1MB	200KB
AOD	10KB	10TB
tag data	100B	200KB

3.3.1.1 Produzione dei dati negli esperimenti HEP

Per fissare le idee su quali sono i tipi di dati che una griglia deve gestire, come sono prodotti, organizzati, distribuiti ed usati, prenderemo in considerazione l'ambito della fisica delle alte energie (High Energy Physic) e gli esperimenti ad essa legati. Altri settori quali l'osservazione del pianeta (Earth Observation), la biologia e la medicina possono essere facilmente ricondotti a questo esempio.

Abbiamo già visto che con l'entrata in funzione del LHC partiranno quattro nuovi esperimenti di HEP: ALICE, ATLAS, CMS e LHCb, che porteranno alla produzione di Petabyte di nuovi dati da analizzare ogni anno.

Possiamo distinguere questi dati in relazione alla loro grandezza e alla frequenza con cui vengono prodotti. Il primo tipo da considerare riguarda i dati prodotti dal detector, il rilevatore di particelle prodotte dagli scontri che avvengono nell'acceleratore. Questi dati sono chiamati eventi e vengono prodotti ad una frequenza di circa 40MHz. Gli eventi vengono poi filtrati per ridurre la quantità dei dati da memorizzare. Dal filtraggio escono dati con una frequenza di circa 100Hz, che prendono il nome di raw data, e ciascuno di essi occupa circa 1MB. Questi vengono poi elaborati con specifici algoritmi per produrre i reconsructed data, di circa 100-500KB ciascuno. I tag data invece sono dati che servono a riassumere informazioni presenti nei raw data e nei reconstructed data, di circa 100 bytes. Nei vari documenti pubblicati si possono trovare anche i termini ESD (Event Summary Data) e AOD (Analysis Object Data), che sono due categorie di dati che nascono da una suddivisione dei reconstructed data, i primi occupano circa 200KB, gli altri invece circa 20KB.

Una importante caratteristica dei dati in gioco è quella di essere principalmente read-only.

3.3.1.2 Use Cases per applicazioni HEP

Vediamo come DataGrid può essere utilizzato nell'ambito dell'HEP. Le attività che manager e fisici dovranno svolgere sulla griglia sono di tre tipi:

- Testing di algoritmi: quella di testare un nuovo codice è una delle operazioni principali svolte dalla griglia sia nelle fasi iniziali di progetto che durante il suo funzionamento. I dati di ingresso e di uscita non sono molto grandi e devono essere localizzati a priori. Dal punto di vista delle operazioni svolte dalla griglia questo compito è abbastanza semplice.

- Produzione di dati: questa operazione è un po' come una simulazione del rivelatore che entrerà in funzione con l'LHC. Per testare l'efficienza della griglia su grandi quantità di dati, quelle che saranno prodotte dal LHC, c'è bisogno di testare il funzionamento dell'intero sistema con dati di prova.

- Analisi dei dati: consiste nell'applicare, all'insieme di dati (eventi) prodotti dall'LHC, in maniera iterativa, dei *cut predicate,* ovvero degli algoritmi di selezione che, in base ai valori di alcune proprietà, scartano gli eventi meno importanti, restringendo il set dei dati di input per le iterazioni successive. Man mano che le iterazioni vanno avanti la quantità di dati da analizzare diminuisce, ma la loro "qualità" aumenta. In questa fase, l'applicazione di un cut predicate ad un insieme di dati di ingresso è eseguito, da parte dall'applicazione utente della griglia, *submitting a job to the DataGrid,* ovvero specificando un pezzo di codice contenente l'eseguibile o un riferimento ad esso, l'ubicazione dei dati di ingresso e quella dei dati di uscita, e dando in pasto questa specifica alla griglia.

3.3.2 DataGrid e le applicazioni Bio-Mediche

Le recenti scoperte avvenute riguardo il genoma umano hanno aperto un vastissimo campo di ricerca per le applicazioni bio-mediche. Queste scoperte hanno inevitabilmente posto una grande pressione sui laboratori di ricerca, principalmente dal punto di vista delle risorse da impiegare per gestire una quantità di dati di qualche ordine di grandezza superiore ai precedenti. Si richiedono inoltre nuove tecniche di data-sharing, data-mining e di gestione delle immagini. Inoltre, anche se non direttamente coinvolte nelle fasi di testbed o nelle prime applicazioni, saranno sicuramente beneficiati dallo sviluppo della tecnologia Grid quei settori medici che vanno di pari passo con la ricerca nucleare. Alcuni esempi riguardano l'elettronica e le tecniche di rivelazione che vengono usate negli esami per la diagnosi precoce di determinati tipi di tumori.

3.3.2.1 Use Cases per le applicazioni biomediche

L'uso di una Griglia da parte delle applicazioni biomediche può essere analizzato considerando separatamente due specifici campi applicativi: quello della ricerca post-genomica e quello dell'elaborazione delle immagini mediche.

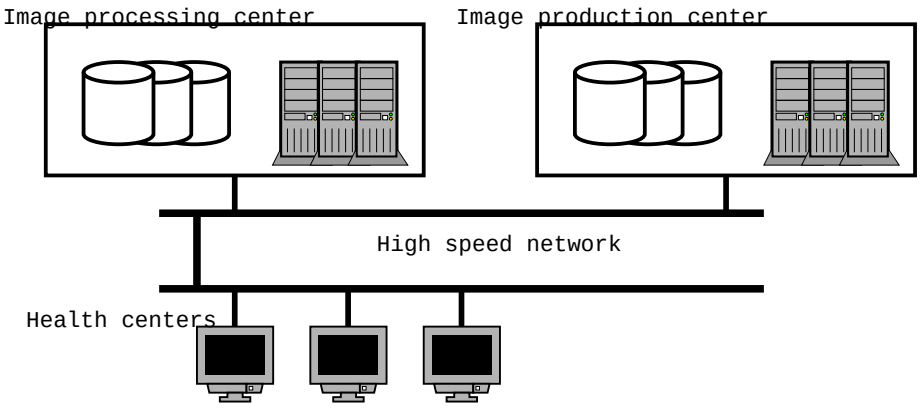


Figura 3.6: Schema architetturale per l'accesso e l'elaborazione remota di immagini mediche

Ricerca genomica Oggi i biologi si trovano ad affontare una crescita esponenziale dei database che riguardano il genoma umano e i sempre più numerosi esperimenti legati ad esso. La situazione attuale pone dei grossi ostacoli al mondo della ricerca a causa della scarso coordinamento dei vari laboratori e dei provider che offrono i servizi di immagazzinamento e analisi dei dati. Gli aggiornamenti sulle sequenze di DNA avvengono giornalmente e i database raddoppiano le loro dimensioni circa ogni 18 mesi. Inoltre si suppone che ad oggi solo il 40% delle sequenze conosciute sia presente nei database. I centri che si occupano di immagazzinare ed offrire pubblicamente l'accesso ai dati non sono molti, e sono tra loro indipendenti, rendendo i risultati di una ricerca variabili in relazione al DB usato. A causa di questa mancanza di coordinamento anche il formato dei vari database può variare, variando con esso la modalità di accesso ai dati, e rendendo impossibile una ricerca che coinvolge più di un database. Il principale strumento di analisi usato dai biologi si chiama BLAST (Basic Local Alignment Search Tool) ed è un programma che, data sequenza di DNA o di aminoacido, effettua una ricerca all'interno di un database per travere delle omologie tra la sequenza fornita e quelle già memorizzate.

3.3.2.2 Elaborazione delle immagini mediche

Per quanto riguarda il campo dell'elaborazione delle immagini mediche, il problema principale è quello dell'acquisizione, quindi del trasferimento tra vari siti, di file di immagini che possono raggiungere anche l'ordine dei GB. Una architettura Grid dovrebbe permettere ai medici, dovunque essi siano, di spedire una immagine ad un centro di elaborazione e di ricevere i risultati di tale elaborazione in breve tempo, magari pochi minuti.

Dati i sempre più numerosi metodi di acquisizione delle immagini, si pensa che un centro di radiologia possa produrre annualmente anche una decina di TB di dati. Anche in

questo settore l'anarchia dei metodi di acquisizione, immagazzinamento e trasmissione pone un serio freno allo sviluppo della ricerca, anche se, ultimamente, il formato DICOM [inserire biblio] è uno standard generalmente accettato, ed un buon candidato a descrivere i file di immagini mediche memorizzate nella Griglia. Alcuni algoritmi di elaborazione delle immagini richiedono ingenti risorse hardware. La possibilità di suddividere le varie elaborazioni e svolgerle in maniera parallela su diversi siti potrebbe apportare dei notevoli vantaggi. Alcune applicazioni, magari riguardanti operazioni chirurgiche, dovrebbero avere la priorità rispetto ad altri task, e poter sfruttare appieno risorse hardware e risorse di rete.

3.3.3 DataGrid e l'Earth Observation

Mentre la cosiddetta Earth Science copre una vasta gamma di applicazioni, incluse quelle sullo studio del clima, la meteorologia, la geologia, lo studio dell'ambiente, dell'agricoltura, etc., l'Earth Observation si rivolge in maniera specifica alla raccolta e all'analisi di dati provenienti da sistemi di rilevamento remoti, finalizzati allo studio della superficie terrestre, terra, ghiacci, mari e atmosfera. Questi sistemi di rilevamento, satelliti e sonde spaziali, inviano dati che vengono raccolti da stazioni terrestri che li rendono disponibili per ulteriori elaborazioni. I componenti principali di questa architettura sono controllati da agenzie spaziali nazionali ed internazionali (ESA, NASA, NOAA).

3.3.3.1 Produzioni dei dati nelle applicazioni di EO

I dati raccolti ed elaborati si possono suddividere in 4 livelli:

Livello 0 Raw data provenienti dai rilevatori spaziali e raccolti presso le stazioni terrestri

Livello 1 Dati di livello superiore ottenuti tramite elaborazioni radiometriche e geometriche

Livello 2 Dati geofisici standard. Possono essere considerati i dati di base per le applicazioni di Earth Science e provengono da un singolo rilevatore

Livello 3 Dati prodotti solitamente da centri di ricerca scientifici e compagnie private. Spesso sono ottenuti mettendo insieme dati di Livello 2 provenienti da diverse sorgenti; rilevatori, GIS[5], simulatori, etc.

[5]Geographical Information Systems

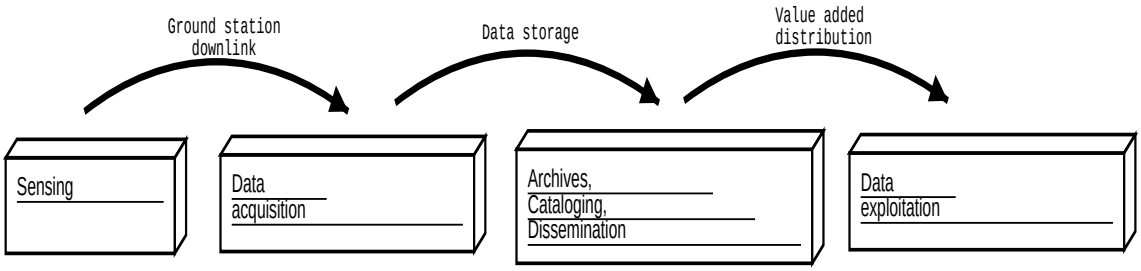


Figura 3.7: Catena di produzione dei dati usati nelle applicazioni di EO

Livello 4 Sono i dati di livello più alto e derivano dalla elaborazioni di dati di livello 3. Tra questi troviamo i risultati delle analisi meteorologiche, dati sulle mappe geografiche, etc.

Dato il grande numero di strumenti, tipi di dati ed applicazioni usate è abbastanza difficile stabilire la grandezza di questi dati, che in genere varia da pochi Kilobytes ad alcuni Gigabytes. In figura 3.7 è schematizzata la catena di produzione dei dati

Diversi componenti concorrono alla formazione di una infrastruttura per applicazioni di EO:

- un efficiente sistema di comunicazione
- archivi e sistemi di acquisizione
- dati, di cui prima abbiamo visto la classificazione
- parametri e metadati che caratterizzano gruppi di dati secondo: il sistema di acquisizione e le sue proprietà, il tipo di dato (livello), la data di acquisizione e il trattamento che ha subito, regione di interesse, qualità, formato etc.
- applicazioni, che spesso costituiscono una interfaccia verso sistemi di calcolo paralleli ad alte prestazioni. Queste applicazioni vengono sviluppate usando sistemi standard distribuiti object oriented come CORBA e XML
- algoritmi e modelli di sviluppo per effettuare le trasformazioni sui dati
- interfacce e sistemi di visualizzazione

3.3.3.2 Esempi di applicazioni di EO

Un primo esempio di applicazione di EO che potrebbe fruire delle risorse messe a disposizione da una griglia, riguarda l'esperimento GOME (Global Ozone Monitoring Experi-

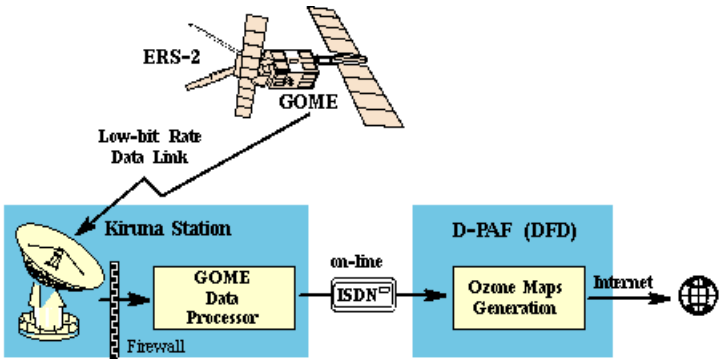


Figura 3.8: Rilevamento e produzione delle mappe nell'esperimento GOME

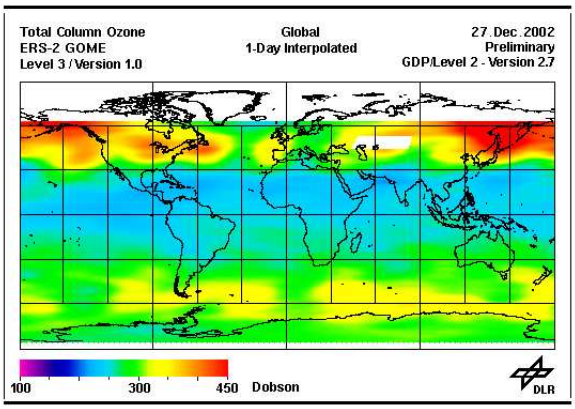


Figura 3.9: Esempio di mappa prodotto dall'esperimento GOME

ment [7], lanciato nel 1996 a bordo del satellite ERS-2 per effettuare misure dell'atmosfera terrestre e in particolare del profilo della distribuzione di ozono. I dati provenienti dal satellite vengono registrati presso alcune stazioni presenti in Svezia, Canada e Spagna e spediti periodicamente al DFD (German Remote Data Sensing Center) che si preoccupa di generare vari tipi di mappe pubblicate in Internet. Questo processo di sviluppo è riassunto in figura 3.8, mentre in figura 3.9 è mostrato un esempio di mappa prodotta tramite le elaborazioni eseguite al DFD.

Altre applicazioni potrebbero essere quelle che utilizzano il modello RAMS (Regional Atmospheric Modeling System), un sistema sviluppato all'Università del Colorado ed usato in tutto il mondo per simulare e prevedere gli eventi atmosferici.

Abbiamo visto qualche esempio dei campi applicativi che potrebbero usufruire dei servizi messi a disposizione da una Griglia, e che oggi collaborano alla costruzione di applicazioni Grid. Tutte queste applicazioni hanno in comune la necessità di raccogliere dati da apposite sorgenti o rilevatori, gestirli in maniera coordinata e sicura, manipolarli con

algoritmi e renderli disponibili ai centri coinvolti nel relativo progetto.

3.4 MONARC, il modello a centri regionali

La memorizzazione dei dati e tutte le elaborazioni che devono essere fatte su di essi, portano alla discussione di come devono essere distribuite, anche geograficamente, le risorse. Bisogna creare una struttura che offra la disponibilità (on site) di dati e risorse di calcolo a numerosi centri di ricerca, università ed aziende sparse per il mondo che lavorano su un determinato progetto.

Nel progetto EU DataGrid e negli esperimenti legati al LHC viene utilizzato un modello di distribuzione delle risorse basato sui Centri Regionali, che sfrutta il lavoro fatto nell'ambito del progetto MONARC (Model Of Networked Analysis At Regional Centres, [32]). Il modello proposto è un modello gerarchico, composto da vari *tiers* [6] in cui il CERN si colloca al primo livello, o tier0. Qui risiede quella parte della struttura di calcolo e acquisizione dei dati che non può essere replicata altrove, come l'archivio dei raw data e i database ESD e AOD. Queste risorse devono essere rese disponibili ai livelli sottostanti tramite reti ad alta velocità. Seguono i tier 1,2,3 e 4 che hanno una decrescente complessità e capacità operativa

I tier 1, nell'ordine di qualche unità, situati ad esempio in Francia, Gran Bretagna, Italia, USA, Giappone rappresentano centri regionali che devono fornire mezzi di calcolo e dati, in maniera complementare a quelli offerti dal CERN, in modo da consentire a numerose persone di lavorare al progetto pur essendo al di fuori della struttura del CERN. I centri tier 2 sono costituiti da Università ed Istituti di ricerca che ospitano l'utenza finale. I tier 3 e 4 sono costituiti da piccole risorse di calcolo o di storage temporaneo, ad esempio una piccola rete di Istituto per quanto riguarda il tier 3, o quello dei desktop per il tier 4, ove ogni ricercatore esegue le proprie elaborazioni dovendo accedere alle innumerevoli risorse di calcolo e di dati messe a disposizione dai nodi superiori della struttura. L'accesso deve essere semplice, veloce, sicuro e coordinato.

Questa struttura gerarchica può essere estesa e contenere più strati. Per semplicità, in seguito faremo riferimento ad una struttura con 4 livelli (0,1,2,3).

[6] dall'inlgese tier=fila

Capitolo 4

Consistenza dei dati replicati

In questo capitolo introdurremo il meccanismo della replicazione, un metodo molto utilizzato nelle applicazioni distribuite per aumentare le prestazioni e la tolleranza ai guasti. Questa tecnica, allorché usata per replicare dati aggiornabili, introduce il problema della consistenza delle repliche. Esistono vari approcci per risolvere questo problema; non esiste un metodo migliore degli altri in generale, poiché ognuno ha le sue caratteristiche ed il suo campo di applicazione. Risulta quindi fondamentale uno studio dettagliato sui requisiti imposti dall'ambiente di utilizzo, attraverso il quale è possibile individuare il modo migliore per risolvere il proprio caso. Vedremo inoltre le tecniche utilizzate in alcuni dei più famosi DBMS (commerciali e non) per risolvere il problema della consistenza in un sistema replicato.

4.1 Introduzione

Il problema delle repliche è un argomento che è stato già analizzato sia nel campo dei sistemi distribuiti che in quello dei database[1], per aumentare le prestazioni e la tolleranza ai guasti. I meccanismi ed i protocolli usati in queste due discipline sono simili ma non identici. Un sistema distribuito può essere modellato come un insieme di servizi forniti dai server ed usati dai client, quindi ciò che viene replicato è un fornitore di servizio (server) con il suo stato. Nei database invece vengono replicati dati (record, tabelle o interi database), ovvero le informazioni immagazzinate o parte di esse. Si capisce che il nostro problema è del secondo tipo, o meglio la teoria dei database è quella che ci può fornire più suggerimenti.

[1] Si parlerà di database e non di database distribuiti per non creare confusione, é ovvio peró che la teoria dei database comprende quella dei database distribuiti, ed é questa quella che piú ci interessa.

Nei database, così come nelle griglie, l'obiettivo è quello di accedere ai dati localmente per diminuire il tempo di risposta ed eliminare l'overhead[2] dovuto alla comunicazione con siti remoti. Infatti, se un utente deve accedere a un'informazione presente nel sistema, si capisce bene che più questa informazione è distribuita nel sistema (più copie identiche, repliche, presenti nei vari nodi del sistema), maggiori sono le prestazioni dell'intero sistema, dal momento che ogni utente può accedere direttamente al dato che più gli conviene (generalmente quello più vicino). In un sistema centralizzato invece, dove le informazioni sono disponibili solo in una sede centrale, le prestazioni sono minori dal momento che tutti gli utenti devono far riferimento alla sede centrale per ottenere l'informazione desiderata, e la sede stessa, dovendo soddisfare tutte le richieste degli utenti, non può garantire un basso tempo di risposta. Questo problema aumenta all'aumentare delle dimensioni del sistema (numero di nodi).

Un sistema centralizzato inoltre costituisce il cosiddetto "single point of failure", ovvero il sistema è meno tollerante ai guasti dal momento che, se la disponibilità della sede centrale che ospita i dati viene a mancare anche temporaneamente, gli utenti non hanno modo di accedere alle informazioni di cui necessitano.

I meccanismi di replica ben si adattano a questi scopi specialmente se le operazioni eseguite sui dati sono in gran parte di lettura. Operazioni di scrittura sui dati introducono il problema della consistenza tra repliche. Infatti, se un utente può modificare una replica, le altre si vengono a trovare in uno stato inconsistente poiché non contengono le ultime modifiche. È quindi possibile che un secondo utente, volendo accedere lo stesso dato ma attraverso un'altra replica, possa leggere delle informazioni che non sono aggiornate in quanto non contengono le modifiche fatte dal primo utente. Sono necessari quindi dei protocolli e dei meccanismi per fare in modo che l'aggiornamento effettuato su una replica, dopo un tempo variabile a seconda dei metodi utilizzati, sia disponibile su tutte le altre repliche del sistema.

Vedremo quali sono i diversi approcci al problema e le caratteristiche di ognuno. La prima grande distinzione da fare per definire i diversi approcci al problema della consistenza delle repliche è quella tra metodi Eager e metodi Lazy.

4.2 Eager Replication

I metodi di Eager Replication vengono anche detti *sincroni* poiché tutte le repliche vengono aggiornate all'interno di una singola transazione. Quando un utente modifica una replica di un dato, prima che questa modifica sia accettata dal sistema, viene propagata a tutte le altre repliche dello stesso dato (*update propagation*). In questo modo abbiamo la

[2] Overhead inteso sia in termini di tempo, ossia l'incremento del tempo di risposta, sia in termini di traffico aggiuntivo immesso nella rete.

certezza che, in ogni momento, tutte le repliche sono perfettamente *sincronizzate*, ovvero contengono esattamente le stesse informazioni.

Questi metodi, vengono anche chiamati metodi *pessimisti* in quanto risolvono il problema drasticamente[3], senza lasciare scampo alla consistenza. Se da un lato offrono le migliori garanzie, i metodi Eager hanno i loro difetti e i loro punti deboli. I difetti più grandi sono quelli di aumentare il tempo di risposta di una operazione di aggiornamento e di diminuire le prestazioni dell'intero sistema generando un alto traffico di dati, nelle fasi di update propagation, non sempre necessario.

I metodi Eager inoltre mal si adattano agli ambienti dove i nodi possono essere temporaneamente disconnessi. Infatti, se un nodo per qualche motivo viene a trovarsi sconnesso dalla rete, un eventuale operazione di aggiornamento che coinvolgerebbe una sua replica viene ritardata in maniera indefinita, fino a quando il nodo non ritorna disponibile. Il problema dei nodi disconnessi può essere risolto tramite dei meccanismi di voting[28]: questi tentano di risolvere il problema cercando di stabilire se, e in quali casi, questo nodo temporaneamente indisponibile ha la capacità di ritardare tutta la propagazione dell'aggiornamento. Ad ogni nodo del sistema viene assegnato un certo valore di importanza. La fase di update propagation viene fatta precedere da un meccanismo di *voting* in cui le repliche coinvolte segnalano la propria disponibilità insieme con il loro grado di importanza. A questo punto il sistema decide se la propagazione dell'aggiornamento può avvenire o meno, in base al raggiungimento di un *quorum* tra le repliche disponibili. In caso positivo, eventuali nodi sconnessi avranno l'obbligo, una volta ritornati disponibili, di aggiornare le proprie repliche. Per non permettere l'accesso a dati *stale* (non aggiornati), le repliche ospitate da un nodo sconnesso dalla rete non devono essere accessibili. In questo modo possiamo far dipendere il buon esito di una operazione di aggiornamento in base alla disponibilità di un certo numero di repliche, che può essere a maggioranza o meno in relazione al valore di importanza assegnato precedentemente ad ogni sito.

L'implementazione di metodi Eager fa leva su meccanismi di lock per prevenire potenziali anomalie (inconsistenze) e convertirle in attese. Le repliche coinvolte in una operazione di aggiornamento sono infatti inaccessibili per tutto il corso dell'operazione. Tentativi di accedere queste repliche vengono bloccati e ritardati in maniera dipendente dalla specifica implementazione.

4.3 Lazy Replication

I metodi di Lazy Replication sono meno rigidi e permettono ad alcune repliche, per un certo periodo di tempo, di trovarsi in uno stato inconsistente[4]. La fase di aggiornamento

[3] Eager infatti si può tradurre con "impaziente" o "impetuoso"

[4] Lazy si può tradurre con "lento" o "pigro"

della singola replica e quella di propagazione dell'aggiornamento avvengono all'interno di due transazioni separate. Questo fa sì che il tempo di risposta del sistema, inteso come tempo necessario all'aggiornamento di una singola replica, diminuisca. Questi metodi sono anche detti *ottimisti* in quanto "sperano" che le repliche non vengano usate nell'intervallo di tempo in cui sono inconsistenti. La bontà di questa ipotesi varia con alcune caratteristiche del sistema, prima fra tutti la frequenza di accesso alle repliche da parte degli utenti. Come vedremo nelle prossime sezioni ci sono vari modi di implementare un metodo Lazy e di variare l'intervallo di tempo di inconsistenza di una replica.

I metodi Lazy si offrono anche come alternativa ai meccanismi di voting per risolvere il problema dei nodi sconnessi. In questi metodi oltre ad aumentare il tempo di risposta del sistema, aumenta anche la disponibilità delle varie repliche, essendo minore il tempo in cui esse sono bloccate dal sistema. Un altro vantaggio è quello della maggiore autonomia dei siti e della flessibilità dei meccanismi di implementazione. Tutte queste proprietà fanno sì che questi metodi siano i più utilizzati in ambienti con reti poco affidabili. Alcuni di questi concetti saranno più chiari con la lettura della prossima sezione, che effettua una classificazione precisa in base alle varie caratteristiche del sistema.

4.3.1 Tassonomia dei metodi Lazy

Una prima classificazione, come riportata in [33], può essere fatta in base a tre parametri:

- *dove* un aggiornamento può essere fatto: update transfer model
- *cosa* è trasferito come un aggiornamento: unit of transfer
- *chi* trasferisce l'aggiornamento: direction of transfer

4.3.1.1 Update Transfer Model

Il primo parametro ci consente di distinguere tra metodi *single-master* e metodi *multi-master*. Nei sistemi single-master una replica è designata come master. La scelta viene fatta in fase di inizializzazione del sistema. La replica master è quella che contiene sempre il valore aggiornato; ogni aggiornamento viene fatto prima sulla replica master e dopo viene propagato alle altre repliche. Questi sistemi hanno il pregio della semplicità: essendo gli aggiornamenti diretti sempre alla replica master questa è in grado di risolvere eventuali conflitti e di stabilire il giusto ordinamento tra gli aggiornamenti. Di contro questi sistemi, essendo in un certo senso centralizzati, reintroducono i classici problemi relativi al "collo di bottiglia" presente in prossimità del master e del "single point of failure". Il tempo di risposta in questi sistemi è minore rispetto ai sistemi multi-master. In questi ultimi infatti l'aggiornamento può essere fatto su più repliche, al limite su ogni replica

nei sistemi *update anywhere-anytime* dove ogni replica può agire da master; sarà compito dei vari master risolvere eventuali conflitti e fare un *merge* dei vari aggiornamenti. Questi sistemi sono molto complessi ed hanno il problema della perdita degli aggiornamenti[5]. La risoluzione dei conflitti e il merging degli aggiornamenti dipende molto dal tipo di applicazione, dal contenuto dei dati e dalle modalità di aggiornamento: una grossa semplificazione è introdotta in quesi sistemi dove le informazioni presenti in un dato possono essere semplicemente aggiunte, senza problemi di ordinamento. Nei sistemi eager questa classificazione ha poco senso, dato che non esiste mai una replica più aggiornata delle altre.

4.3.1.2 Unit of transfer

Una modifica ad un dato può essere espressa sia in termini del suo contenuto, sia in termini di quali operazioni sono state eseguite su di esso. Questo secondo parametro distingue tra metodi *contents-transfer* e metodi *log-transfer*. Nei primi viene trasferito il contenuto del dato, e possiamo distinguere tra *total file replacements*, dove il nuovo dato sostituisce per intero quello vecchio e *binary difference* dove invece viene trasferita solo la modifica al file, calcolata ad esempio con strumenti analoghi al comando *diff* dei sistemi Unix. Nei metodi log-transfer invece ciò che viene trasferita è una sequenza di comandi per modificare il dato; questo metodo ben si adatta ai sistemi in cui i dati, come singole entità, possono avere una grande dimensione e in cui le modalità di aggiornamento sono ben codificate. Un semplice esempio di questi dati sono i file che contengono database, che vengono aggiornati con comandi SQL. Questo parametro può essere utilizzato indistintamente anche nei metodi Eager.

4.3.1.3 Direction of transfer

Questo parametro determina quale replica è incaricata di effettuare il trasferimento dell'update, variandone nel contempo il momento di inizio. In questo caso distinguiamo tra sistemi *pull-based* e sistemi *push-based*. Nei primi ogni replica è incaricata di controllare gli aggiornamenti di cui necessita, rivolgendosi direttamente (*polling*) ad una replica master in caso di sistemi single-master o multi-master o ad una qualsiasi replica in caso di sistemi update anywhere-anytime. In questo caso una replica non corre mai il rischio di ricevere lo stesso aggiornamento più volte, essendo essa stessa a richiedere le informazioni di cui necessita. L'intervallo di polling è stabilito in base ai requisiti dell'applicazione fino al limite in cui una replica richiede gli aggiornamenti solo nel momento in cui viene usata. Dall'altra parte ci sono i sistemi push-based, in cui è la replica che contiene degli

[5] Infatti i conflitti sono rilevati e risolti *dopo* che un aggiornamento è stato accettato e magari l'applicazione utente che lo ha effettuato si è già sconnesso dal sistema

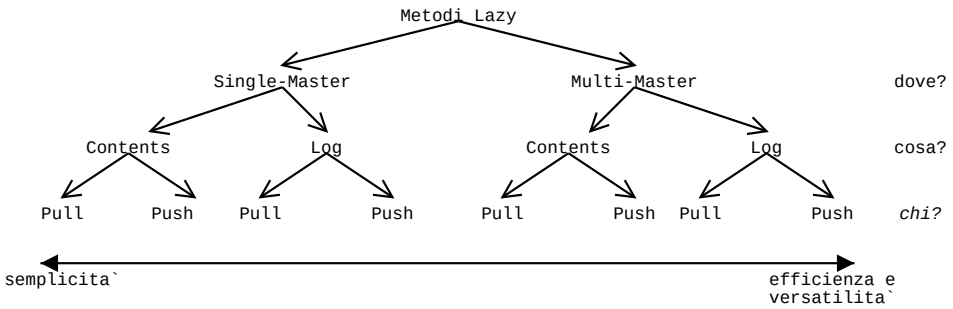


Figura 4.1: Tassonomia dei metodi di Lazy Replication

aggiornamenti ad essere incaricata di traferire tali aggiornamenti a tutte le altre repliche. Questi sistemi possono ridurre al minimo l'intervallo di tempo in cui una replica si trova in uno stato inconsistente ed eliminano l'overhead dovuto al polling. Di contro, nei sistemi push, può accadere che una replica riceva degli aggiornamenti duplicati. Nei sistemi Eager è implicito l'utilizzo del metodo push.

In figura 4.1 è rappresentato lo schema dell'intera classificazione effettuata.

4.3.2 Requisiti di consistenza

Per definizione, i metodi di replicazione Lazy non possono assicurare un livello di consistenza stretto, che è invece prerogativa dei metodi Eager. Sarebbe sbagliato comunque pensare che tutti i metodi Lazy offrono le stesse garanzie in termini di consistenza. Si tratta quindi di fissare dei nuovi criteri in base al quale poter stabilire delle diverse "garanzie" di consistenza.

4.3.2.1 Eventual Consistency

Costituisce il requisito minimo di ogni algoritmo di replicazione. Richiede che, qualunque sia lo stato delle repliche di un sistema, se non avvengono ulteriori aggiornamenti e le repliche possono comunicare liberamente per un tempo sufficiente, alla fine lo stato delle repliche converga ad una situazione consistente, in cui tutte le repliche siano identiche. Si capisce che un algoritmo che non soddisfa l'Eventual Consistency risulta praticamente inutilizzabile. Il problema del mantenimento della Eventual Consistency può essere suddiviso in tre parti:

1. distribuire gli aggiornamenti tra le repliche
2. determinare l'ordine di applicazione degli aggiornamenti

3. rilevare e risolvere eventuali conflitti tra gli aggiornamenti

Un modo per ottenere facilmente la Eventual Consistency può poggiare su un sistema single-master pull-based. Per stabilire l'ordine tra gli aggiornamenti e risolvere eventuali conflitti viene solitamente usato un *timestamp* che accompagna ogni replica ed indica il tempo in cui il suo contenuto ha subito l'ultima modifica. Questo lavoro spetta alla replica master che può utilizzare le comuni tecniche di controllo della concorrenza dei DBMS come *two-phase locking* e *optimistic concurrency control* [31] e [19]. Nei sistemi multi-master le cose si complicano poiché più aggiornamenti possono essere effettuati contemporaneamente su diverse repliche e possono raggiungere altre repliche in ordine casuale. Per risolvere questo problema si ricorre solitamente all'utilizzo dei *timestamp-vector*. Un timestamp-vector è un vettore che accompagna ogni replica master e riassume lo stato delle altre repliche master. Le dimensioni del vettore crescono all'aumentare del numero dei master in gioco, e questo pone un problema non indifferente per quanto concerne la scalabilità. Di fatto l'utilizzo di questa tecnica in grandi sistemi come può essere una Griglia, dove il numero delle repliche può raggiungere alcune migliaia, è praticamente impossibile se il numero di master supera una certa soglia. Il modo in cui i master comunicano per risolvere eventuali situazioni di conflitto dipende strettamente dalla semantica del sistema e dei dati in gioco.

4.3.2.2 View Consistency

La Eventual Consistency, di per sé, fornisce poche garanzie all'applicazione utente finale riguardo alla qualità dei dati che egli utilizza. Lo scopo della View Consistency è proprio quello di controllare la qualità utilizzati da un utente intervevendo direttamente sulle richieste di lettura. All'interno della Eventual Consistency possiamo distinguere tra *Causal Consistency* e *Bounded Inconsistency*.

La prima preserva un ordinamento tra gli accessi ad una replica fornendo diverse garanzie tra cui possiamo citare la *read your writes*, in cui la lettura di un utente è sempre in grado di vedere le modifiche fatte precedentemente dallo stesso utente, sulla stessa replica, e la *monotonic reads*, in cui successive letture da parte di un utente sulla stessa replica resituiscono sempre dei risultati con livello di qualità (in termini di consistenza) crescente. *Writes follow reads* e *monotonic writes* sono altri due requisiti il cui significato può essere facilmente dedotto dai precedenti. Ad ogni modo, per raggiungere queste garanzie in ambienti multi-master, è necessario l'utilizzo dei timestamp vector già visti, che possono asscurare un livello di ordinamento parziale o totale tra i vari aggiornamenti (fare riferimento a [33] per approfondimenti)

La Bounded Inconsistency permette all'applicazione utente di specificare il massimo grado di inconsistenza che può essere tollerato su una replica, specificato per esempio da un

intervallo di tempo oppure da una differenza tra numeri di versione. Questo approccio è abbastanza semplice, e può essere ottenuto, in un sistema single-master, con un periodico meccanismo di push o pull degli aggiornamenti.

4.4 Il problema della scalabilità

L'abilità del sistema a scalare rispetto al numero delle repliche in gioco è una caratteristica molto importante dei sistemi di replicazione. All'aumentare del numero delle repliche infatti il tempo di propagazione degli aggiornamenti aumenta. Questo problema è maggiormente presente nei sistemi single-master, in cui l'unico master presente deve trasferire tutti gli aggiornamenti alle altre repliche. In questo caso, organizzare le repliche in una struttura ad albero può rendere il sistema più efficiente. Un altro modo per ridurre il tempo di update propagation è l'utilizzo di tecniche di trasferimento multicast.

All'aumentare del numero delle repliche aumentano i conflitti tra gli aggiornamenti. In [29] si stabilisce che i sistemi multimaster sono molto più sensibili rispetto a quelli single master[6]. All'aumentare del numero delle repliche aumenta inoltre lo spazio occupato dalle informazioni di stato usate per gestire la replicazione e il mantenimento della consistenza. Queste informazioni possono essere quelle presenti nei cataloghi delle repliche o quelle dei timestamp vector.

Infine, all'aumentare delle repliche, aumenta il traffico di rete generato per mantenere le repliche consistenti. I protocolli utilizzati devono quindi tenere conto della bontà, sia in termini di banda che di affidabilità, della rete su cui poggia il sistema.

4.5 Meccanismi di replica usati nei DBMS

4.5.1 Objectivity/DB

Questo ODBMS include la funzionalità DRO (Data Replication Option) che fornisce un metodo di replica trasparente alla applicazione utente. Objectivity/DRO utilizza un metodo sincrono con un meccanismo di voting per consentire una modifica di una immagine del database. In base a questo, ogni accesso che intende modificare l'immagine locale del DB viene portata a termine solo se si raggiunge un quorum tra le immagini attualmente disponibili. Se ciò avviene, la modifica si compie con successo su ogni replica disponibile ed ogni eventuale replica del DB che era down, quando ritornerà attiva,

[6] Se n è il numero delle repliche, nei sistemi multi master i conflitti sono $O(n^2)$ mentre in quelli single master solo $O(n)$.

verrà automaticamente aggiornata allo stato del quorum (la maggioranza delle immagini, o repliche).

Il quorum non necessariamente deve coincidere con la maggioranza. Possiamo infatti fare in modo che una immagine del DB sia considerata master semplicemente variando il suo peso, e ponendolo maggiore della metà delle altre immagini (che hanno peso unitario). In questo modo, per ottenere il quorum, l'immagine master deve essere sempre disponibile. Aggiustando il peso di ogni immagine, Objectivity/DRO supporta diverse configurazioni da master/slave a peer-to-peer passando per configurazioni miste. Il metodo di gestione delle repliche in Objectivity non è comunque pensato per funzionare attraverso WANs senza un netto calo delle prestazioni.

4.5.2 Oracle

Oracle9*i* affronta il problema delle repliche fornendo, come altri DBMS, diversi meccanismi sui quali l'applicazione utente può agire per ottenere le migliori prestazioni in relazione alle sue specifiche necessità. L'utente ha la possibilità di definire tabelle master e snapshot, le seconde sono delle copie (istantanee) ospitate presso altri siti non master che servono appunto per implementare il meccanismo di replica.

4.5.2.1 Asynchronous multi-master

In questo modello si suppone di effettuare il processo di replica tra n siti master coinvolgendo intere tabelle master. Le master table di ogni sito possono essere aggiornate ed i cambiamenti vengono automaticamente, in modo asincrono, propagati agli altri siti tramite RPCs, che possono essere schedulate per eseguire il processo di sincronizzazione quando meglio si crede. Se due siti tentano di aggiornare le medesime informazioni all'interno del medesimo intervallo di sincronizzazione si può creare un conflitto. Oracle9*i* permette all'applicazione utente di scegliere in che modo risolvere i conflitti, usando i timestamp oppure le priorità, per dare maggior autorevolezza ad alcuni master piuttosto che ad altri.

4.5.2.2 Materialized Views

Per andare incontro a diversi requisiti come il supporto per configurazioni gerarchiche, la possibilità per un sito replica di ospitare solo parti di tabelle e la possibilità di un sito di effettuare la sincronizzazione on-demand e in maniera rapida, Oracle9*i* mette a disposizione le materialized views (MVs), o snapshot. Queste possono contenere una copia intera o parziale di una master table e vengono aggiornate dal master periodicamente oppure on-demand. Esiste inoltre la possibilità di aggiornare direttamente una updatable MV e di

propagare i cambiamenti alla MV master, che può essere una master table oppure un'altra MV. In questo modo si possono creare strutture gerachiche per distribuire le informazioni in maniera molto efficiente. Per andare incontro alle esigenze dei mobile users invece il processo di sincronizzazione delle MVs è stato ottimizzato per supportare il trasferimento di grandi moli di dati su reti ad alta latenza (WANs or Internet).

4.5.2.3 Hybrid Solutions

I primi due meccanismi possono essere facilmente combinati per ottenere diversi modelli. Oltre ai metodi visti Oracle9*i* offre due ulteriori meccanismi di replica, la procedural replication e la sinchronous replication. Il primo non è un vero e proprio meccanismo di replica, ma raggiunge gli stessi effetti. Infatti, invece di replicare i dati, replica la procedura di modifica dei dati. In alcuni casi infatti, su scala periodica, vengono eseguite delle operazioni che coinvolgono una grande quantità di dati. Può essere conveniente, anzichè replicare tutti i dati, passare la procedura eseguita da un sito ad un altro, ed eseguire la procedura localmente. Il secondo metodo è il classico metodo sincrono, utile nel caso in cui le repliche non siano numerose e siano collegate tra di loro da una rete a bassa latenza.

Oracle9*i* offre inoltre il Replication Management Tool, un modo semplice e user-friendly di organizzare e mantenere il processo di replica da un unica postazione di lavoro. Tramite un'interfaccia grafica si può agire sugli attori coinvolti nella replica, sui dati replicati e sui metodi utilizzati.

4.5.3 IBM DB2

La soluzione offerta da IBM è composta da 4 componenti principali: Administration, Capture, Apply e Alert. Questi quattro componenti interagiscono tramite le control tables, tabelle in cui vengono memorizzati dati che servono al funzionamento del sistema. I suddetti programmi girano su dei logical server [7]; ci possono essere tre tipi di logical server: source server, target server e control server. Il source server contiene il meccanismo di change-capture, le source tables che si vogliono replicare e le control tables per il programma Capture. Il target server contiene le target tables, destinazioni per quanto riguarda la direzione degli aggiornamenti. Ciò non implica necessariamente una politica master-slave, dove le tabelle slave siano di sola lettura; la distinzione source-target come vedremo sfuma in relazione alla configurazione del sistema di replica. Il control server contiene le control tables per il programma Apply. Vediamo il funzionamento dei quattro componenti:

[7] In questo caso il termine Server si riferisce semplicemente ad un database, e non ad un Server nel senso del modello Client-Server

- Administration: è la sede di comando del sistema di replica, dotata di una GUI (replication center), che permette di pianificare e successivamente gestire il corretto funzionamento della replicazione. Inizialmente vengono definite le sorgenti del sistema di replica e quindi vengono associate alle destinazioni. Questo componente deve avere un collegamento con i componenti Capture e Apply e può girare su sistemi Windows e Unix/Linux.

- Capture: le modifiche apportate alle source table vengono registrate in file di log, e da qui catturate, appena dopo la fase di commit, da questo programma che gira in un source server . Una volta catturate queste modifiche vengono temporaneamente memorizzate in apposite tabelle (CD, change data) utilizzate dal meccanismo di propagazione degli update, ed infine cancellate una volta completato l'aggiornamento del/dei targets. A seconda della configurazione del sistema (master-slave, update anywhere...) istanze dei programmi Capture e Apply devono girare su uno o più server logici.

- Apply: le modifiche catturate nella fase precedente vengono applicate alle target tables da questo programma, che può girare su qualunque server ma deve avere una connessione sia con il source che con il target. Il programma Apply, così come quello di Capture, possono essere schedulati per entrare in funzione quando meglio si crede, variando così la "rilassatezza" degli aggiornamenti e con sé il livello di consistenza offerto. Ad ogni modo, dato che il programma Capture rende disponibile l'aggiornamento una volta eseguito il commit della modifica locale, la replicazione non può che essere asincrona. Inoltre il metodo di propagazione degli update è generalmente di tipo pull, ovvero è il programma Apply che contatta il source server per ricavare le informazioni di aggiornamento.

- Monitor: questo programma, tramite le sue control tables definite durante il processo di configurazione, monitorizza il funzionamento del processo di replica interagendo con le control tables presenti nel source e nel target server. Si possono definire delle soglie oppure degli eventi generati dai programmi Capture e Apply da segnalare agli utenti finali tramite e-mail. Anche il funzionamento di questo programma può essere schedulato per controllare la situazioni a intervalli regolari o a orari definiti.

4.5.3.1 Scenari previsti

Nella documentazione disponibile sul sito IBM [30] vengono riportati alcuni scenari tipici che possono aiutare l'applicazione utente a scegliere il sistema di replica a lui più congeniale. Vediamole brevemente:

- Data distribution and data consolidation: in questo scenario sono presenti una o più sorgenti di dati (source server) accessibili in lettura/scrittura e una o più destinazioni (target server) accessibili solo in lettura. Il modello è quello classico di una replica di dati read-only.

- Bidirectional exchange of data with a master: pensando di avere una sorgente e più destinazioni, le modifiche possono essere fatte direttamente su un target server, ma in questo caso vengono poi propagate agli altri target passando dal source server. È il caso tipico di un modello master-slave (o primary copy)

- Bidirectional with no master (peer-to-peer): in questo scenario si evitano i problemi legati al possibile collo di bottiglia creato da un singolo master. Adesso gli aggiornamenti effettuati localmente vengono direttamente passati agli altri peer systems senza necessariamente passare da un singolo server. Questa soluzione fornisce inoltre bilanciamento del carico, alta disponibiltà e resistenza ai guasti.

4.5.4 Sybase

Il sistema di replica fornito da Sybase comprende un insieme di Replication Server che collaborano per scambiarsi le informazioni che raccolgono ognuno dalla propria sorgente di dati. Ogni Replication Server è responsabile per uno specifico database ed i vari database possono anche non risiedere sulla stessa LAN. Le modifiche apportate ad un database vengono rilevate dal Replication Server tramite analisi del file di log, che contiene i record per quelle transazioni che hanno già superato la fase di commit.

Gli scenari previsti sono i medesimi visti per gli altri DBMS, quindi andiamo direttamente al caso che più ci interessa, quello in cui un utente intende modificare una copia locale dei dati. Sybase Replication Server Environment raccomanda che ogni dato abbia un preciso sito che lo controlla vestendo i panni del possessore di tale dato. Per la modifica di un dato non gestito dal server locale, Sybase offre due soluzioni:

- Asincrona: l'applicazione utente rivolge la richiesta di update al server remoto tramite una comunicazione asincrona (RPC). Una volta accettata la richiesta è il server remoto che apporta la modifica al database che egli stesso controlla, e che si incarica di propagare gli update agli altri server, compreso quello che ha iniziato la richiesta.

- Sincrona: la seconda alternativa consiste nel "loggarsi" all'interno del server remoto, effettuare gli update in modo sincrono, dopodichè, come nel caso precedente, le informazioni vengono propagate agli altri server, compreso quello richiedente.

Sybase sconsiglia l'uso di una soluzione del tipo update-anywhere, dove i dati appartengono a tutti, perché questa soluzione renderebbe molto più complicata la fase di risoluzione dei conflitti, di back-up e di recovery. Inoltre una soluzione di questo tipo porta inevitabilmente a dei compromessi per quanto riguarda il livello di consistenza offerto, è difficile identificare il vero valore di un dato in un certo istante. Si legge inoltre nella documentazione ufficiale [35], che una eventuale politica di risoluzione dei conflitti si troverebbe ad affrontare i problemi legati alla sincronizzazione degli orologi di sistema e quelli dovuti alla latenza nelle comunicazioni. Quindi, sebbene niente nell'architettura del Replication Server impedisca agli utenti di progettare e mettere su un ambiente del tipo update-anywhere con tanto di sistema di risoluzione dei conflitti, Sybase raccomanda, per ottenere un ambiente più gestibile, l'utilizzo di una primary copy.

4.5.5 MySQL

Dalla versione 3.23.15 MySQL supporta il meccanismo di replica distinguendo tra server master e slave. Il server master è quello a cui devono essere indirizzate tutte le query che apportano modifiche al DB mentre operazioni di lettura possono essere indirizzate a server slave. I server slave ospitano una replica del DBMS (dalla versione 3.23.16 si può restringere la replica a un set di database) e si collegano periodicamente (o in seguito ad una caduta) al server master per aggiornare il loro stato. Il server master, infatti, è incaricato di tenere un file di log in cui memorizza le modifiche apportate al DB da un certo istante in poi, e di ricevere le connessioni da parte dei server slave. MySQL al momento supporta solo un master e più slave. Nelle future versioni saranno aggiunte politiche di quorum per variare dinamicamente il master e meccanismi per effettuare il load balancing delle richieste di lettura tra differenti slave.

4.5.6 Microsoft SQL Server 7.0

Il DBMS targato Microsoft offre un meccanismo di replica che si basa sul modello "publish and subscribe". Questo modello può essere studiato analizzando le proprietà e le capacità di ogni entità che vi partecipa.

- Publisher: è il server che rende i dati disponibili per la replica e che si occupa della loro gestione. Ogni "data element" ha un singolo publisher che si occupa di memorizzare le informazioni relative a quel dato, specialmente quelle di aggiornamento.
- Subscribers: sono i server che ospitano alcune repliche e ricevono le informazioni di update.

- Publication: è un insieme di articoli. Gli articoli contengono gruppi di dati che devono essere replicati, possono contenere “righe” o “colonne” (frammentazione orizzontale e verticale).

Ci sono due metodi di subscription, push e pull. Nel primo è il server publisher che si occupa di propagare gli update verso i subscribers. Nel secondo invece sono questi ultimi che richiedono gli aggiornamenti. Questi due metodi forniscono differenti livelli per quanto riguarda alcune proprietà di un metodo di replica quali la sicurezza, l’indipendenza dei siti, la consistenza etc.

4.5.6.1 Snapshot Replication

E’ la soluzione più semplice disponibile nel caso in cui l’applicazione presenti i seguenti requisiti: basso volume di dati replicati, aggiornamenti poco frequenti, basso grado di consistenza e grande autonomia tra publisher e subscribers. In questa soluzione il publisher esegue una istantanea dei dati da replicare e li manda periodicamente ai subscribers. Da notare che ciò che viene trasferito non sono le informazioni di update dei dati, ma i dati stessi. Se un sito che ospita le repliche è momentaneamente down, riceverà i nuovi dati al ciclo successivo.

4.5.6.2 Transactional Replication

Questo metodo offre un livello di consistenza alto derivante dal fatto che gli aggiornamenti ai dati devono essere necessariamente effettuati dal publisher, che poi si occupa di trasferirli ai subscriber in maniera più o meno “near real time”. Gli aggiornamenti spediti sono le transazioni che hanno passato lo stato di commit.

Entrambi i metodi hanno la possibilità di abilitare l’opzione di “Immediate Updating Subscriber”. Questa offre la possibilità di effettuare degli update dal lato subscriber site, che effettua una modifica alla propria copia locale, modifica che tramite two-phase commit protocol (2PC) viene riflessa immediatamente al publisher site. Se la modifica ha buon esito, il publisher si occupa di trasmetterla ai rimanenti subcribers, in maniera adeguata al grado di consistenza richiesto.

4.5.6.3 Merge Replication

Merge Replication offre il più alto grado di autonomia tra siti. In questa soluzione Publisher e Subscribers possono aggiornare le loro copie in maniera indipendente l’uno dall’altro, per poi mettere insieme il risultato delle loro operazioni (merging) con una periodicità

variabile da qualche ora a diversi giorni. Durante il merging possono sorgere dei conflitti sui dati aggiornati da più parti in gioco: questa situazione viene automaticamente risolta sulla base di priorità assegnabili ai siti o in base al timestamp delle operazioni.

4.5.7 Differenze con l'ambiente Griglia

Il problema della replicazione e della consistenza dei dati in una Griglia non può essere risolto sfruttando i meccanismi classici che abbiamo appena visto per i DBMS per una serie di motivi. In una Griglia abbiamo a che fare con sistemi eterogenei: i dati e le applicazioni usate per elaborarli sono molteplici. Questa eterogeneità rende impraticabile l'uso di metodi che dipendono fortemente dalla semantica dei dati, come l'uso di tecniche di aggiornamento di un file tramite trasferimento di comandi (vedi sezione 4.3.1.2). Non solo, ma questa forte eterogeneità che caratterizza il sistema Griglia ci impone di gestire il problema della consistenza in maniera molto flessibile per l'applicazione utente: infatti, soluzioni valide per un certo tipo di applicazioni e di utenza potrebbero risultare impraticabili per altri utenti. È consigliabile quindi che il futuro Servizio di Consistenza delle repliche permetta all'utente di scegliere la politica che più si adatta al suo caso. In una Griglia inoltre il numero elevatissimo di repliche da gestire, la scala di diffusione geografica e i requisiti di scalabilità nel tempo costituiscono dei vincoli che non possono essere rispettati dalle soluzioni che abbiamo appena visto per i DBMS. Al momento non sappiamo nulla riguardo a soluzioni al problema della consistenza in altri progetti Grid, e questo ci fa capire ancora meglio la complessità del problema che stiamo trattando.

Capitolo 5

Data Management in EDG WP2

Questo capitolo ci introduce con maggiore dettaglio nelle questioni che riguardano il WP2 di EDG, ovvero il Data Management. Cercheremo di fissare i punti chiave dell'attività del WP2 con maggior attenzione agli aspetti che riguardano la replicazione. Vedremo come questo meccanismo era supportato nella release 1 di EDG e quali miglioramenti sono arrivati con la release 2. Questo ci permetterà di introdurre la terminologia ed i concetti che utilizzeremo nel capitolo successivo.

5.1 Task Area del WP2

Il Data Management è un concetto molto ampio, alcuni dei task che spettano al WP2 hanno una natura ben definita, altri invece sono di più difficile comprensione e definizione, e richiedono una stretta collaborazione con altri work package, WP1 e WP3 in particolare. I lavori del WP2 sono stati inizialmente divisi in 5 settori (task area), che sono:

1. Data Acces & Migration
2. Data Replication
3. Meta Data Management
4. Secure & Trasparent Data Access
5. Query Optimisation

Questi cinque task hanno una complessità variabile, e sono stati numerati appunto a partire dai più "semplici".

5.1.1 Granularità dei dati

Inizialmente si è pensato di stabilire il file come l'unità di dati su cui lavorare. Questa limitazione può essere accettabile, ma ben presto si è sentita l'esigenza di trattare gruppi di file o *collezioni*, così come sarebbe auspicabile ridurre la granularità a livello di *oggetti*.. Un file infatti può essere composto da più oggetti, e magari ogni oggetto può aver una sua precisa semantica: all'interno di una analisi di HEP, un file potrebbe contenere varie misure relative a diversi eventi, così ogni oggetto potrebbe raccogliere un insieme di misure effettuate su un singolo evento. Una granularità ridotta a livello di oggetto potrebbe rivelarsi utile nei casi in cui si voglia trasferire solo parte dell'informazione contenuta in un file. A tale scopo un Grid Object Management è in fase di studio nell'Optimization Task.

5.1.2 Sistema di replicazione

Uno degli obiettivi centrali del WP2 è quello di fornire all'applicazione utente finale un accesso globale ed efficiente ai dati presenti nella Griglia. Vedremo che questo può essere fatto creando, dove opportuno, alcune repliche dei dati. La gestione di un sistema di repliche è una cosa complessa, che deve comprendere un catalogo (Replica Catalog), un sistema di gestione delle repliche (Replica Manager) ed altri servizi, come quello di Replica Selection ed il Consistency Service, di cui parleremo in dettaglio nei prossimi capitoli.

5.1.3 Memorizzazione dei metadati

Per assicurare il buon funzionamento dei servizi del Grid Middleware, si deve tenere conto anche dei *metadati*, ovvero dei dati interni utilizzati dai servizi del WP2 ed altri. Questi riguardano statistiche, dati degli utenti, dati di monitoraggio del sistema ed altri, esclusivamente per uso "interno". Per raccoglierli e renderli utilizzabili è stato progettato l'SQLDatabase service.

5.1.4 Ottimizzazione dell'accesso ai dati

Ottimizzare l'accesso ai dati è una delle questioni principali che spettano al WP2. Noi vedremo in particolare i meccanismi di ottimizzazione legati all'uso della replicazione.

5.2 Data Replication

Nelle prime versioni di EDG, release 1.4 compresa, tutti i file (repliche) all'interno della griglia erano trattati allo stesso modo. In seguito è stato suggerito l'uso di due tipi di file: *master* file e repliche. Il master file è gestito dal suo proprietario, mentre le repliche fanno parte della griglia e sono gestite in maniera opportuna del Grid Middleware e dai suoi sotto-servizi. In questo senso la parola master ha un significato diverso da quello che abbiamo visto a proposito dei protocolli di consistenza. L'uso delle repliche deve essere trasparente all'applicazione utente finale; queste devono essere create, copiate ed eliminate all'occorrenza, allo scopo di migliorare l'accesso ai dati degli utenti. Inizialmente le repliche dei file erano per definizione read-only, questo ne facilitava la gestione ed evitava i problemi di sincronizzazione e consistenza di cui ci dovremo occupare.

5.2.1 Nomi dei file, attributi e lifetime

Le convenzioni sui nomi dei file sono state, e lo sono tutt'ora, oggetto di discussione all'interno del progetto EDG. Le prime implementazioni del software EDG usavano tre tipologie di nomi:

- Logical File Name: è identificato da una stringa che deve rispettare la URL-like syntax [RFC 2396], per esempio:
  *"lfn://eo.esa.int/anything+:you*like.tex"*
  La prima parte, eo.esa.int, (virtual hostname) non si riferisce al dominio in cui risiede il file; l'informazione contenuta nel Logical File Name quindi non riguarda né la locazione del file né come vi si può accedere.

- Physical File Name: è usato per identificare in maniera univoca dove un file è fisicamente memorizzato. Adottando la stessa sintassi [RFC 2396], un pfn è una stringa del tipo:
  "pfn://cms.cern.ch/Grid/daq/triggers/2001/challenge02/ev001"

  dove cms.cern.ch indica l'hostname dell'SE dove il file è memorizzato, seguito dal path opportuno.

- Transport File Name: il tfn è usato per indicare come si può accedere un file e contiene informazioni sufficienti per permettere ad un client di recuperarlo. Un esempio di tfn può essere:
  "http://cms001.cern.ch/Grid/daq/ev001"
  oppure

“Gridftp://kinky.cern.ch/anything/you/like/ev001”

Spesso la traduzione pfn/tfn è fatta direttamente dal grid middleware, quindi l’applicazione utente può non preoccuparsi del protocollo di rete utilizzato per il trasferimento.

Questa convenzione sui nomi è ancora oggi oggetto di discussioni all’interno del progetto EU DataGrid, specialmente per il fatto che il pfn identifica univocamente il file solo se questo risiede su disco. In realtà, il sistema di memorizzazione dei file prevede che il disco sia usato praticamente come cache, mentre il supporto vero e proprio è costituito dai Mass Storage System, che hanno una capacità molto più grande. Per questo, nel primo prototipo di RM (Marzo 2002), è stata adottata una convenzione che usa:

- FileName = host name + percorso del file su disco:
  host1.cern.ch/data/run1/file1.dbf

- Logical FileName. Si ritiene valida la definizione data in [15], per cui:
  lsfn = “lfn://” + “virtual hostname” + “/” + “anystring”:
  *lfn://eo.esa.int/anything+:you*like.tex.*

Ancora una volta è bene precisare che l’informazione contenuta nel logical file name non dà nessuna indicazione su dove risiede il file né su come vi si accede. Il lfn, che è tutt’ora oggetto di discussioni, deve essere unico all’interno di un RC di una VO e può seguire dei vincoli imposti da ogni VO. Il FileName (che corrisponde al vecchio pfn) invece indica la posizione di un physical file, quindi contiene l’indicazione dello SE su cui risiede. Affinchè sia valido, un FileName, deve contenere un SE registrato presso il Replica Catalog Service.

Per quanto riguarda le copie master e secondary, in [38] si stabilisce che, dato un set di repliche identiche, solo una può avere l’attributo di master, le altre sono copie secondarie, e questi attributi vengono memorizzati nel Replica Metadata Catalog (RepMeC).

Un’altra precisazione da fare riguarda gli attributi dei file. Ce ne possono essere diversi, e possono risiedere sia nel Replica Catalog che nel Replica Metadata Catalog. In particolare, nel RC, vengono memorizzati gli attributi associati ai file attualmente presenti nello SE, mentre nel RepMeC si trovano quelli associati al lfn che sono indipendenti dalla locazione di una replica. Alcuni attributi possono essere presenti in entrambi i cataloghi per aumentare la robustezza del servizio e la capacità di rilevare le inconsistenze. In ogni caso, per accedere a questi attributi è necessario passare dal RM per assicurare la consistenza delle informazioni.

In relazione al life time di un file su un SE, si possono distinguere 3 tipi di file: permanenti, durabili e volatili. Un file “master” deve essere sempre del primo tipo in quanto è il punto di riferimento per garantire la consistenza delle repliche. Una copia secondaria invece può essere di uno qualsiasi dei 3 tipi.

5.2.2 Identificatori unici e alias

Per gestire correttamente file e collezioni all’interno di una VO è necessario che questi siano identificabili in maniera univoca. D’altra parte, utenti diversi potrebbero voler identificare uno stesso dataset, ciascuno con nomi diversi. Per soddisfare entrambi i requisiti è stata proposta l’adozione di uno schema che prevede un *GUID (Grid Unique IDentifier)* per ogni dataset, a cui possono essere collegati con una relazione 1..n più *lfn alias*, ovvero dei nomi più leggibili del GUID, personalizzabili e modificabili. Il GUID è assegnato ad ogni dataset al momento della creazione o della registrazione e segue uno schema ben preciso che lo rende unico all’interno di una VO. Il GUID associato ad ogni lfn è trattato come un attributo e immagazzinato nel RLS. Ovviamente la relazione tra GUID e lfn impone dei vincoli di consistenza, ad esempio cancellando un lfn il GUID non deve essere rimosso, e la cancellazione di un GUID rende inutilizzabili gli lfn collegati ad esso. Il RM si preoccuperà di tutte le possibili situazioni critiche (vedere [24] per approfondimenti).

5.2.3 Collezioni

Le collezioni sono insiemi di file (e/o collezioni stesse) che vengono raggruppati in base a proprietà comuni (contenuto, data di creazione, sito di appartenenza etc.) per facilitarne la gestione. Le collezioni possono essere di due tipi: *confined collections* e *free collections*. Le prime sono simili ad archivi tar o zip, e contengono file che vengono trattati come un unico file, residenti in unico SE. Le seconde pongono meno limiti, i file possono essere liberamente aggiunti o tolti dall’insieme, ma non garantiscono che gli elementi della collezione siano validi e accessibili in un dato istante. Nel seguito il termine dataset farà riferimento a un file o una collezione.

5.2.4 Replica Manager

Il Replica Manager (RM) è il servizio che si occupa della creazione delle repliche, del loro spostamento e della loro cancellazione. Attraverso il Replica Catalog service, con cui deve interagire in maniera consistente, il RM è a conoscenza dell’esistenza e dell’esatta locazione di tutte le repliche. Il RM deve inoltre usare un servizio di trasferimento file, come GridFTP [9], per effettuare i suddetti spostamenti. Deve inoltre fare i conti con

le varie policy relative all'accesso ai file e alla loro modifica, che possono cambiare in base agli utenti e alle VO. In un primo momento il RM doveva trattare solo dati read-only, in seguito si è reso necessario lo studio di un Consistency Service. Un altro servizio molto importante all'interno del Replica management è quello di Replica Selection & Cost Estimation, che deve affrontare le problematiche dell'individuazione delle repliche e in particolare della *"best replica"*. In base a diversi criteri (distanza, banda della rete, carico, throughput etc..) una replica può essere migliore o peggiore di un'altra per un eventuale accesso o trasferimento.

5.3 EDG release 1

I componenti del software EDG relativi alla replicazione hanno subito considerevoli modifiche nel passaggio dalla release 1 alla 2. In questa sezione cercheremo di riassumere brevemente i componenti della release 1.

5.3.1 Globus Replica Catalog

Fino alla release 1.4 inclusa, il Replica Catalog service era unico e centralizzato per ogni VO, basato sulla implementazione di Globus[5], che usava LDAP come sistema di gestione del catalogo.

5.3.2 GDMP

Nelle prime versioni di EDG, il Replica Manager era un servizio costruito attorno al Catalogo delle Repliche Globus e il processo di gestione delle repliche doveva essere controllato direttamente dall'applicazione utente finale. Il tool principale per registrare e replicare file era GDMP (Grid Data Mirroring Package), che usava una architettura client-server e forniva il supporto per molteplici VO. Nelle versioni successive il software di GDMP sarà inglobato nel RM.

5.3.3 Problemi e lacune della release 1

Possiamo riassumere che, nella release 1 di EDG, il sistema di gestione delle repliche era sufficiente ad eseguire i primi testbed ma non era molto flessibile. L'utilizzo di un catalogo centralizzato poneva dei seri problemi di scalabilità e il sistema di gestione delle repliche non disponeva di funzionalità avanzate come quelle di ottimizzazione e di gestione (creazione e rimozione) automatica delle repliche.

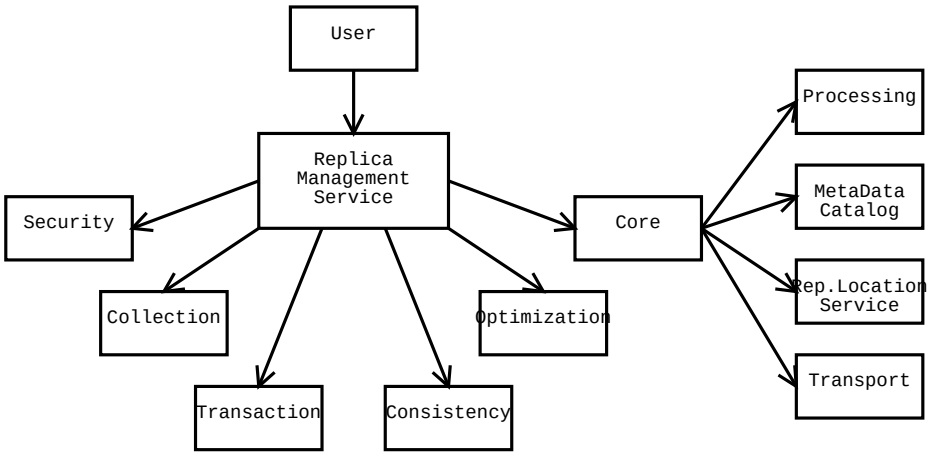


Figura 5.1: Architettura di Reptor

5.4 EDG release 2

5.4.1 Replica Management Service: Reptor

Il Replica Manager (RM) o Replica Management Service (RMS) è il servizio incaricato di assolvere i compiti di gestione delle repliche costituendo il punto di accesso per l'applicazione utente finale e per le applicazioni di middleware. Il RM serve da interfaccia verso i vari sotto-servizi, nascondendone le implementazioni, semplificando l'accesso e la gestione dei dati.

In questa sede faremo riferimento a Reptor[24], il modulo di Replica Manager introdotto nella release 2 di EDG. Il RM può essere configurato come un servizio distribuito, sia per ragioni di performance che di tolleranza ai guasti.

Un passo importante per garantire la consistenza dei dati disponibili su una griglia è quello di controllare e standardizzare il modo in cui vi si accede. Una volta che un file è stato registrato presso il RM, quest'ultimo ne assume il pieno controllo. Eventuali accessi, modifiche, copie etc. devono essere fatte tramite il RM, fino al momento in cui il file viene tolto dal suo controllo.

La figura 5.1 mostra lo schema logico ed i blocchi che costituiscono il servizio di Replica Manager. Le funzionalità principali di Reptor vengono messe a disposizione attraverso le *Core API*, con cui si possono creare repliche, eliminarle e catalogarle. Per far questo le Core API accedono ad altri servizi ed API, come:

- le *Processing API*, necessarie per effettuare operazioni di *pre* e *post processing* prima di un trasferimento di file

- il *Transport Service*, che si occupa del trasporto di file implementato attraverso vari protocolli, tra cui GridFTP [16]

- il *Replica Location Service (Giggle)*, per localizzare le repliche. Di questo parleremo in dettaglio nel prossimo capitolo

- il *Metadata Catalog (RepMeC)*, che memorizza informazioni sui file utili al RM: size, checksum, data, tipo, alias, lifetime, master copy, transaction specific metadata. Ci sono poi informazioni che riguardano le collezioni, di cui parleremo più avanti: elements, type. Informazioni che riguardano la sicurezza: proprietario, permessi di accesso , group membership, local policies e access control lists. Inoltre possono essere definiti vari tipi di metadati che sono specificatamente legati all'applicazione che li utilizza.

Secondo questo schema, il Servizio di Consistenza che dovrà essere sviluppato si presenta come un sotto-servizio del RM.

In [24] si stabilisce che il Consistency Service, in relazione al grado di consistenza offerto, è responsabile di:

1. propagare gli aggiornamenti in seguito a modifiche di file. Il metodo usato dipende dalla organizzazione delle repliche (zero, una o più repliche master), dal grado di "rilassamento" permesso e dalle politiche relative al quorum in caso di repliche inaccessibili

2. rilevare le situazioni di inconsistenza. Queste possono essere causate da accessi non autorizzati ad un file, ma anche da situazioni più critiche come guasti e attacchi al sistema e possono comprendere l'utilizzo del checksum di un file per stabilire se la replica è o meno una copia esatta

3. gestione del ciclo di vita delle repliche. Una replica infatti può esistere ed essere "registrata" per un periodo più o meno breve, in relazione al suo utilizzo. Una volta terminato il suo ciclo di vita ed eliminata, devono essere eliminate anche le informazioni nel RC ed ogni riferimento ad essa.

Replica Manager API Elenchiamo brevemente le API messe a disposizione, per adesso, dal RM. Per una descrizione approfondita, parametri e codici di errore si rimanda a [38].

Core API

- registerEntry(LogicalFileName lfn, FileName source): inserisce una nuova replica nel catalogo. Non presuppone nessun trasferimento di file
- copyFile(Filename source, FileName destination, String protocol): effettua un semplice trasferimento di file senza coinvolgere il RC
- copyAndRegisterFile(LogicalFileName lfn, FileName source): unisce le due funzionalità di copia e registrazione dei metodi precedenti
- replicateFile(LogicalFileName lfn, FileName source, FileName destination, String protocol): si differenzia dalla precedente per quanto riguarda le entità coinvolte nel trasferimento, che in questo caso sono due SE, mentre nel precedente una delle due poteva essere anche un CE
- deleteFile(LogicalFileName lfn, FileName source): elimina un file e rimuove l'entry relativa nel RC
- FileName[] listReplicas(LogicalFileName lfn, ComputingElement CE): elenca tutte le repliche relative ad un file logico
- unregisterEntry(LogicalFileName lfn, FileName source): rimuove l'entry di un file fisico dal RC

Metadata Catalogue API

- bool isMaster(LogicalFileName lfn, FileName source): restituisce *true* se il FileName passato per parametro corrisponde ad una replica master
- FileName getMaster(LogicalFileName lfn): restituisce il FileName della replica master
- setMaster(LogicalFileName lfn, FileName master): fissa una certa replica come master di un file logico

Optimization API

- getAccessCost(LogicalFileName[], ComputingElement, protocol[])
- initFilePrefetch(LogicalFileName[], ComputingElement, protocol[])
- cancelFilePrefetch(LogicalFileName[], ComputingElement)

- getBestFile(LogicalFileName[], protocol[]

Il significato di questi metodi sarà spiegato più avanti nella sezione relativa alle funzioni di ottimizzazione 5.5.1.

5.4.2 Replica Location Service: Giggle

Uno dei problemi introdotti dal meccanismo di replica è quello di individuare correttamente l'ubicazione delle varie copie di un determinato file. Dato un lfn, dove posso trovare una sua replica?

Nell'ambito del WP2 è stato sviluppato un servizio, Giggle (GIGa-scale Global Location Engine), che si occupa di affrontare il problema in modo flessibile e altamente configurabile così da poter rispondere alle esigenze di diverse VO.

Consideriamo il caso in cui le repliche vengano ospitate in diversi *replica site*. Ogni replica site ospita dei meccanismi di memorizzazione per le repliche e un *local replica catalog (LRC)* che si occupa delle corrispondenza lfn,pfn per ogni pfn[1] presente nel sito. I vari LRC hanno il compito di spedire informazioni circa il proprio stato a dei *replica location index nodes (RLIs)* che si occupano di gestire le informazioni sullo stato globale delle repliche. Per individuare la posizione di una replica relativa ad un lfn sarà quindi sufficiente effettuare una interrogazione al sistema di indici o direttamente ad un LRC.

Il funzionamento dell'intero servizio può essere descritto tramite sei parametri, configurabili per andare incontro a diverse esigenze.

5.4.2.1 Parametri e logica di funzionamento

Cerchiamo di entrare più nei dettagli dei vari componenti del sistema. Giggle non definisce come deve essere implementato un LRC, quello che serve è che mantenga le associazioni lfn–pfn in maniera consistente con i dati presenti nel sistema di immagazzinamento gestito (uno o più storage devices). Il LRC deve essere in grado di rispondere alle query (quali pfn sono associati ad un lfn e a quale lfn è associato un certo pfn), di fornire un controllo di accesso alle informazioni supportando la GSI [26] ed infine di spedire periodicamente informazioni di stato ai RLIs.

Si capisce che questo sistema non fornisce dei livelli stretti di consistenza, dato che, tra la modifica (creazione, cancellazione) di una replica e l'aggiornamento dello stato del LRC può incorrere del tempo. Per adesso non prendiamo in considerazione ulteriori specifiche, tipo i casi in cui più VO possono condividere risorse con politiche diverse.

[1] A prescindere dalla convenzione sui nomi utilizzata 5.2.1 il pfn indica in questo caso il nome di una replica

Sebbene l'insieme dei vari LRC forniscono una visione consistente e globale circa le repliche, questi non supportano direttamente le query del tipo "dove è situata una replica del file lfn X?". Per questo abbiamo bisogno di una ulteriore struttura, quella degli indici. Questi mantengono informazioni di associazione (lfn,replica site). L'organizzazione e il funzionamento del servizio è regolato da 6 parametri:

1. G: il numero di RLIs totali presenti nel RLS (≥ 1)
2. R: il grado di ridondanza degli indici, ovvero quante copie di ogni RLI vengono contenute nel RLS
3. Pl: la funzione usata per partizionare lo spazio dei nomi lfn nei vari indici. Se nulla, gli indici devono contenere informazioni su ogni replica, altrimenti solo le corrispondenze relative ad una parte di lfn
4. Pr: la funzione usata per partizionare lo spazio dei replica site associati ad un determinato lfn. Se nulla, gli indici devono indicare, per ogni lfn gestito (vedi 3), l'ubicazione di ogni replica presente nel RLS, altrimenti solo una parte
5. C: la funzione usata per comprimere le informazioni di stato che i LRCs mandano agli indici. Ciò può diminuire il flusso di informazioni spedite in rete
6. S; la funzione usata per determinare quali informazioni di stato spedire (complete o parziali). Si può ad esempio scegliere di inviare con una frequenza relativamente bassa lo stato completo di un LRC, e con frequenza maggiore solo degli aggiornamenti

I RLCs ed i RLIs possono variare nel tempo, entrare ed uscire dal gioco in maniera dinamica. Per questo il sistema deve prevedere un servizio di appartenenza per gestire tali cambiamenti. Nel caso di partizionamento dello spazio dei nomi i LRCs devono sapere a chi mandare le proprie informazioni così come , in caso di ingresso od uscita di un RLI ci deve essere un meccanismo per ridistribuire le informazioni degli indici secondo il partizionamento usato. Vedi [23] per ulteriori dettagli.

5.4.2.2 Alcuni esempi pratici

Con ($G = R = 1$, $Pl = Pr = \emptyset$, $S =$ all, $C = \emptyset$) si ha un indice globale centralizzato. Questa è la situazione più semplice da analizzare, semplice da mantenere ma poco flessibile e scalabile. I LRCs mandano periodicamente il loro stato completo all'unico indice presente, in maniera non compressa. L'indice risponde alla query di un client indicando quali LRCs contengono una replica per il lfn richiesto. Con questa informazione

il client può effettuare direttamente una query al LRC scelto per ottenere il corrispondente pfn. Può accadere che la risposta del LRC non sia positiva, se ad esempio la replica non è più presente in quel sito. Questo comunque è un problema non specifico di questa implementazione, ma diretta conseguenza dell'aggiornamento asincrono degli indici. In applicazioni come quelle che riguardano l'HEP, dove si possono raggiungere centinaia di siti replica con 50 milioni di lfn e 500 milioni di pfn, questa soluzione non è praticabile, sia per quanto riguarda la capienza dell'indice, sia per il traffico generato verso di esso, che costituirebbe un sicuro collo di bottiglia.

Una possibile scelta che potrebbe adattarsi a molte situazioni è quella con $G \gg R > 1$, con il solo partizionamento dei nomi lfn tramite una funzione hash e con $S \neq \emptyset$ e $C \neq \emptyset$. Questa scelta offre ridondanza nel caso di indisponibilità di un indice, scalabilità all'aumento delle repliche ed adeguate prestazioni grazie alla spedizione di informazioni parziali e compresse. Un altro vantaggio della ridondanza degli indici si ottiene dal fatto che ogni "copia" di un indice può rispondere alle query, bilanciando opportunamente il carico di lavoro degli indici.

Possiamo poi pensare ad un partizionamento dei nomi lfn basato sulle collezioni *(LC, logical collections)*, con cui si può ridurre notevolmente la grandezza degli indici e il traffico di update degli indici stessi. Un LRI infatti memorizza informazioni del tipo (LC x, Site y). L'utente quindi, che è a conoscenza del partizionamento, sa che deve fare la query ad un certo indice per avere l'indicazione del sito che ospita quel particolare file, appartenente a quella particolare collezione.

Notiamo infine che è possibile creare una implementazione del RLS con gli indici organizzati a livelli gerarchici. In questo modo l'applicazione utente del servizio effettua una query ad un primo livello di indici per sapere non dove si trova un certo lfn, ma a quale indice deve rivolgere la seconda query, che restituirà l'informazione di locazione del file desiderato. In questo modo aumentano notevolmente le possibilità di configurazione del servizio, diminuendo le conoscenze che deve avere un utente circa il partizionamento degli indici.

5.4.2.3 Conseguenze del rilassamento della consistenza degli indici

Il fatto che il RLS non offra uno stretto livello di consistenza ha delle implicazioni sullo sviluppo del nostro servizio. Infatti bisognerà tenere conto del fatto che, nel caso in cui una replica venga modificata, avremo bisogno di propagare l'aggiornamento a *tutte* le altre repliche. Siamo consapevoli però che una interrogazione al RLS può non fornire l'ubicazione di *tutte* le repliche.

5.5 Replica Optimisation Service: Optor

Nella Griglia il caso d'uso più generale a cui possiamo pensare è quello in cui l'applicazione utente immette un job per eseguire alcune operazioni su alcuni dati. Questo lavoro deve essere svolto dalla Griglia nel modo più veloce possibile; per far questo, uno dei punti chiave è l'impiego di una buona strategia di replicazione e di accesso ai dati. Più i dati sono "vicini" al job, più velocemente il job eseguirà le sue operazioni. Le fasi di ottimizzazione sono quelle che richiedono una più stretta collaborazione tra i vari workgroups di EU DataGrid, in particolare tra WP1,WP2,WP3, WP5 e WP7.

Optor, così viene chiamato il Replica Optimisation Service sviluppato dal WP2 Optimisation Group, svolge due importanti compiti nella fase di ottimizzazione:

- determina il CE più conveniente, dal punto di vista dell'accesso ai dati, su cui eseguire un job
- localizza le *best replicas* di ogni lfn che il job usa per svolgere le proprie operazioni.

Due importanti assunti sono stati fatti nella fase di progetto di Optor:

- un job, in esecuzione in un preciso CE, ha un unico *close* SE associato, ovvero un SE che fornisce performance migliori per quanto riguarda l'accesso ai propri dati
- il job in esecuzione non scrive dati sui file, è ammessa solo la lettura.

Dal punto di vista architetturale Optor si presenta come un modulo interno al Replica Manager (Reptor). Tramite Reptor si possono chiamare delle *funzioni di ottimizzazione* implementate in Optor. Queste funzioni svolgono le loro attività in tre diversi momenti: *scheduling, pre-execution e run-time.* L'ambiente di EU DataGrid è molto dinamico: condizioni di traffico nella rete, disponibilità di siti e di risorse possono variare nel tempo. Per questo, le *optimization functions*, pur essendo simili, possono restituire risultati differenti proprio in relazione al diverso momento in cui entrano in azione.

5.5.1 Replica Optimization Functions

Vediamo brevemente le funzioni messe a disposizione da Optor. Per informazioni più dettagliate si rimanda a [37].

Tabella 5.1: Tabella restituita dalla funzione *getAccessCosts()*

	CE1	CE2	...	CE*n*
lfn1	$best\_pfn_{1,1}$	$best\_pfn_{2,1}$	...	$best\_pfn_{n,1}$
lfn2	$best\_pfn_{1,2}$	$best\_pfn_{2,2}$	...	$best\_pfn_{n,2}$
...	...	...	...	...
lfn*m*	$best\_pfn_{1,m}$	$best\_pfn_{2,m}$	...	$best\_pfn_{n,m}$

- *getAccessCosts()*: questa funzione viene chiamata prima che il job vada in esecuzione. Dato un vettore di *n* possibili CE's su cui svolgere il job e uno di *m* possibili lfn che il job deve accedere, questa funzione, per ogni CE, calcola i tempi di accesso a tutte le repliche di ogni lfn indicato, e restituisce una matrice del tipo mostrato nella tabella 5.1

  in cui ogni casella indica la replica del relativo lfn il cui tempo di accesso è il minimo tra tutte le repliche di quel lfn, per quel particolare CE. Si assume che solo le repliche già esistenti vengano prese in considerazione per il calcolo. I risultati forniti riflettono una situazione, dal punto di vista del carico di rete e delle disponibilità delle repliche, che può essere temporanea e quindi sono soggetti a variazioni.

- *getBestFile()*: questa funzione viene chiamata dal job in esecuzione su un preciso CE. Dato un vettore di *n* lfn a cui il job deve accedere, ne viene restituito uno contenente, per ogni lfn, il migliore pfn disponibile. Viene inoltre deciso se alcuni file devono essere replicati verso il close SE o verso qualunque altro SE. Questi risultati sono indipendenti da quelli restituiti dalla funzione precedente, e possono essere diversi, per i motivi già detti.

- initFilePrefetch(): la chiamata alla funzione precedente può innescare alcuni meccanismi di replica, e quindi può bloccare l'esecuzione del job per qualche tempo. La funzione che stiamo esaminando è la versione non bloccante della precedente. Questa esegue più o meno le stesse funzioni, ma registra il suo stato ed i suoi risultati in un catalogo di meta-dati, che verranno in seguito reperiti ed utilizzati dal job. In pratica questa funzione può essere chiamata dopo che il job sia stato schedulato, ma prima che esso vada in esecuzione (pre-execution time), per avvantaggiarsi nel fare alcune operazioni di replica. Dati di ingresso e di uscita sono i medesimi della funzione precedente. Associata a questa funzione c'é ne è un'altra, *cancelFilePrefetch()*, che si occupa di cancellare e ripulire lo stato di una chiamata a *initFilePrefetch()*, nel caso questa, per diversi motivi, non sia più necessaria.

Altre funzioni come *getSECosts()* e *getNetworkCosts()* vengono usate internamente a Optor, ma sono oggetto di lavoro di altri workgroups, rispettivamente WP5 e WP7.

In [37] si possono inoltre trovare alcuni esempi che possono chiarire il procedere del meccanismo di ottimizzazione, attraverso alcuni sequence diagram che coinvolgono Reptor, il Resource Broker ed altri attori.

Capitolo 6

Simulazioni di strategie di replicazione dinamica con OptorSim

In questo capitolo vedremo come è possibile simulare il funzionamento di una Griglia. Più precisamente cercheremo di capire come eseguire una simulazione per studiare gli algoritmi di replicazione dinamica e come questi possono essere valutati e confrontati proprio in base ai risultati forniti dalla simulazione. Nel prossimo capitolo invece vedremo le funzionalità aggiunte al simulatore per rendere possibile lo studio degli algoritmi di consistenza.

6.1 Ambiente di simulazione

Una simulazione è portata avanti pensando ad una architettura semplificata di Griglia, vista come un insieme di siti; ogni sito contiene al più un Computing Element ed al più uno Storage Element. Il Resource Broker è il componente che si incarica di schedulare i job forniti dall'applicazione utente su un CE. Il job in esecuzione su un CE accederà ai dati (file) del proprio SE o di un SE remoto. Il Replica Manager si incarica di gestire l'accesso ai dati, e tramite il Replica Optimizer si decide quando e dove creare o rimuovere repliche. Questa architettura è ben descritta dalla seguente figura 6.1.

Nella simulazione il RB assicura la distribuzione dei job verso i CE's. 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. Per adesso la fase di ottimizzazione entra in gioco solo dopo che un job è stato schedulato su un CE. Un job, o meglio la sua descrizione (tramite il Job Description Language [36] nell'ambiente reale o semplicemente nel file di configurazione per le simulazioni), contiene una lista di lfn a cui accedere

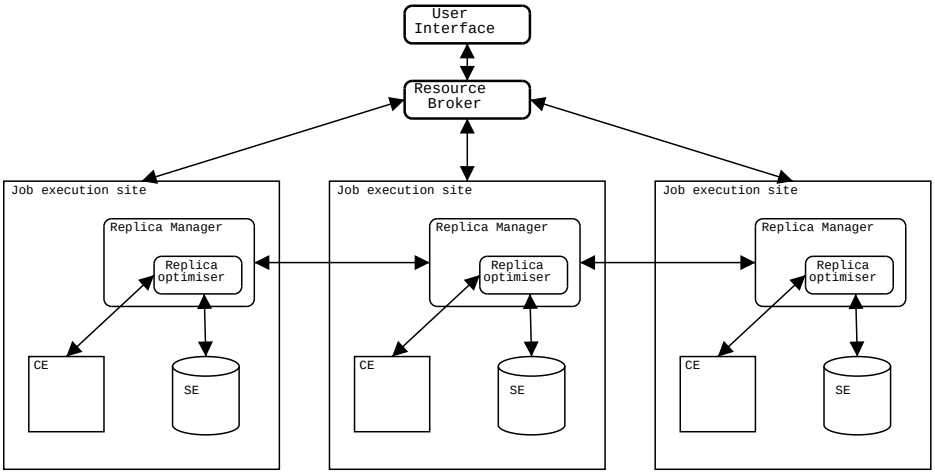


Figura 6.1: Architettura semplificata di una Griglia

per effettuare le proprie elaborazioni. L'ordine in cui si accede a questi file può essere scelto tra diversi modelli: sequenziale, casuale, casuale a passo unitario e casuale a passo Gaussiano, illustrati nella figura 6.2.

Quando un job richiede l'accesso ad un file, il lfn viene usato per reperire, tramite la funzione di ottimizzazione *getBestFile()* (che implementa parzialmente le funzionalità descritte precedentemente), la replica migliore (*best replica*). Il Replica Catalog usato per richiedere informazioni sulle repliche è implementato, all'interno della simulazione, tramite una tabella di lfn e pfn corrispondenti. La *getBestFile()* è una chiamata che può bloccare l'esecuzione del job. Esaminando la banda disponibile tra i vari siti che ospitano le repliche di quel dato lfn e quello locale, ovvero il sito in cui il job verrà eseguito, 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, che può coinvolgere un trasferimento di file (replicazione) oppure un accesso remoto. Non viene simulata la parte che riguarda l'elaborazione dei dati, anche perché poco influisce sul reale scopo della simulazione, cioè quello di analizzare le tecniche di replicazione e il loro impatto sul traffico di rete e sul tempo di esecuzione dei job.

6.2 Algoritmi di ottimizzazione

Come già detto gli algoritmi di ottimizzazione servono per migliorare l'accesso ai dati da parte dei job. Il fatto di poter lavorare sempre su file locali costituisce un notevole vantaggio. Uno dei fattori che distingue i vari algoritmi di ottimizzazione riguarda quindi la scelta sulla creazione di nuove repliche, ma non solo: quando, dopo la chiamata alla

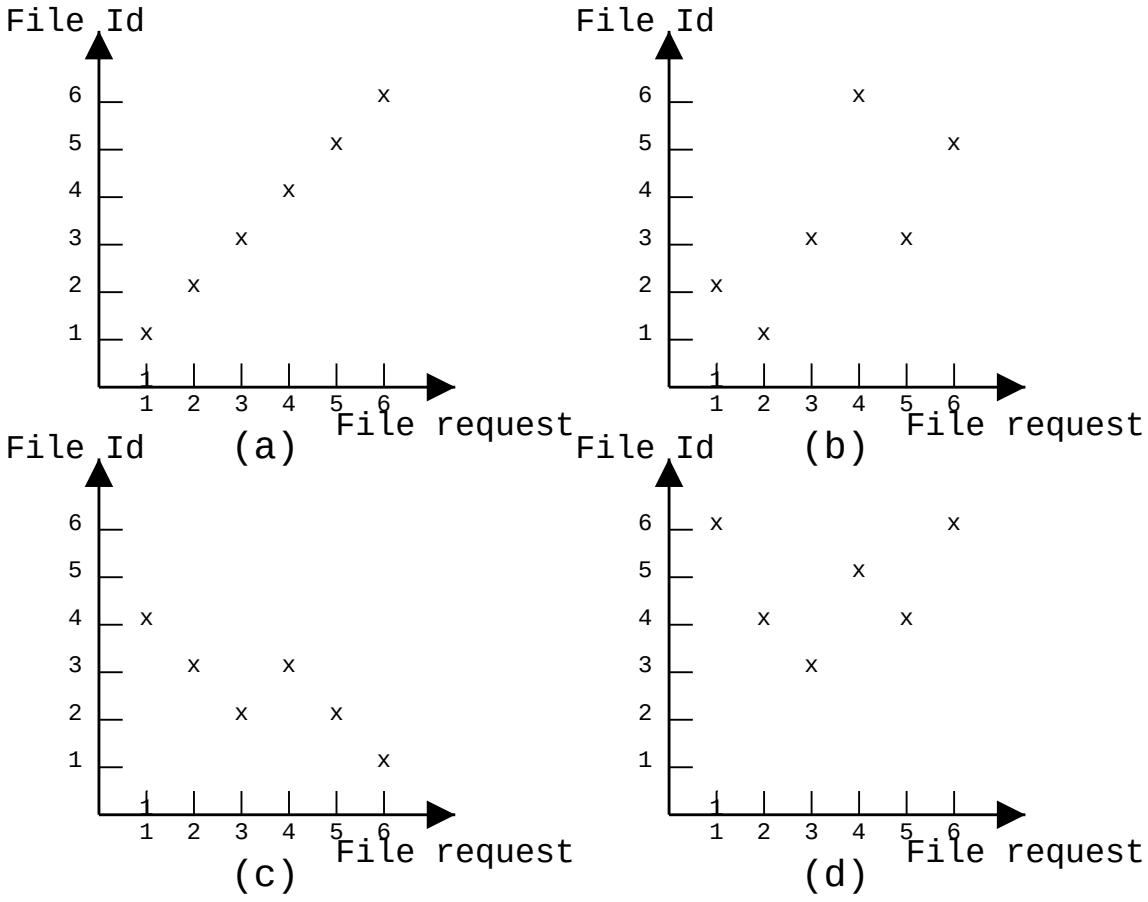


Figura 6.2: Esempi di access patterns: (a) sequenziale, (b) random, (c) random a passo unitario, (d) random a passo Gaussiano

getBestFile(), viene deciso di creare una replica di un dato file sullo SE locale, può succedere che questo sia pieno, e che un'altra replica debba esser rimossa per far posto alla nuova. Per questo, anche la strategia con cui viene scelta la replica da rimuovere può essere diversa. In OptorSim sono stati implementati alcuni di questi algoritmi:

- no replication: in questo caso il posizionamento delle repliche viene deciso all'inizio, e rimane invariato per tutto il corso della simulazione. Se un file richiede l'accesso ad un file la cui replica migliore si trova su un SE remoto, a questa si accederà in maniera remota, senza effettuare nessuna copia locale. Poichè il carico di rete varia durante il corso della simulazione, anche la best replica rifletterà una situazione temporanea (sarà realmmente la "best" solo in alcuni momenti)

- unconditional replication, oldest file deleted: questo algoritmo esegue sempre la replicazione del file nel sito in cui verrà elaborato dal job. Se una replica dovrà essere rimossa, questa sarà la più vecchia all'interno dello SE

- unconditional replication, least accessed file deleted: uguale a prima, solo che la replica rimossa è quella a cui si è fatto accesso da più tempo

- economic model: questo è il modello più complesso. 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. Infatti gli SE potranno rivendere successivamente questi beni ai CE ricavandone un profitto. L'obiettivo dei CE è quello di minimizzare i costi, quello degli SE di massimizzare il profitto. Il valore di un file all'interno di un SE è proporzionale al numero di volte che verrà riferito, e viene valutato mediante calcoli probabilistici. Questo modello viene usato sia per decidere se creare delle nuove repliche, sia per scegliere le eventuali repliche da rimuovere per guadagnare spazio. Approfondimenti sul modello e sui risultati delle simulazioni che lo coinvolgono possono essere trovati in [18] e [17].

6.3 Il Grid Simulator OptorSim

OptorSim è un simulatore di Griglie scritto in Java per la valutazione degli algoritmi di ottimizzazione usati da Optor. Nel corso di questo lavoro vedremo come può essere sfruttato nello studio di un servizio di consistenza. La versione attuale, la 0.5, può essere scaricata all'URL:
`http://edg-wp2.web.cern.ch/edg-wp2/optimization/download.html`
sotto forma di archivio compresso (fare riferimento a [10] per installare ed eseguire il programma).

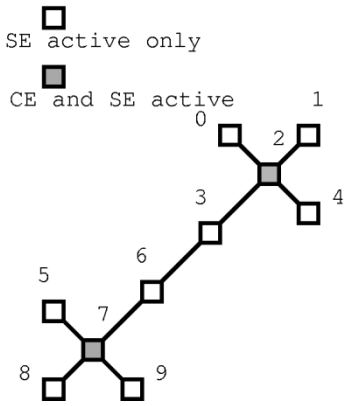


Figura 6.3: Esempio di struttura di una griglia

6.3.1 File di configurazione

OptorSim fa uso di 4 file di configurazione. Il primo, parameters.conf, è quello principale, ma può essere sostituito tramite indicazione di un parametro al momento della chiamata dello script di avvio. Al suo interno vi sono indicazioni sugli altri 3 file.

Il primo di essi è il file di configurazione della griglia all'interno del quale i dati sono organizzati in una matrice. Ogni riga contiene informazioni relative ad un sito: il numero di worker nodes (all'interno dell'unico CE, se presente), il numero di SE (uno o zero) e la sua grandezza. Se queste prime 3 colonne contengono solo zeri allora la riga corrisponde ad un nodo di tipo 'r', network router, ovvero un nodo che non contiene SE's e nemmeno CE's, ma solo dispositivi di rete per lo smistamento del traffico. Le altre colonne contengono la banda di ogni link, espressa in Mb/s, che collega un sito ad un altro (=0 se nessun link è presente). In questo modo si ottiene una descrizione abbastanza semplificata ma chiara della struttura della griglia.

Il secondo è il file di configurazione dei job dove troviamo informazioni relative ai file presenti nella griglia e ai job che verranno eseguiti durante la simulazione. È logicamente suddiviso in quattro parti. La prima riguarda i file presenti nella griglia: nome (lfn unico), grandezza (in MB) e un indice che viene usato nell'economic model, che indica la correlazione tra file e job. Un job richiede con maggior probabilità file con indici vicini. La seconda parte riguarda l'uso dei file da parte dei vari job: per ogni job vengono indicati gli lfn dei file che esso richiederà. La terza parte indica su quali nodi verranno eseguiti i vari job. La quarta ed ultima parte riguarda invece la probabilità di esecuzione di ogni job.

Il terzo file contiene dei dati relativi alla mappa da usare per la simulazione.

Vediamo adesso le più importanti informazioni contenute nel file di configurazione principale. Prima di tutto questo file deve contenere l'indicazione degli altri due file visti

precedentemente, più il numero di job complessivamente eseguiti durante la simulazione (più istanze di un certo job possono venir eseguite nella griglia). Per quanto riguarda la politica dello scheduler, ovvero la scelta del CE su cui eseguire un job, si può scegliere se usarne una casuale oppure fare uso della chiamata alla *getAccessCosts()*. L'algoritmo di ottimizzazione può essere scelto tra quelli indicati nella sezione precedente così come il modello di accesso ai dati. Possiamo inoltre specificare il numero di accessi ai file eseguiti dai job. Se ad esempio fissiamo questo valore a due, ed il job richiede 10 file, si avranno in totale 20 accessi. La distribuzione iniziale dei file può essere scelta tra una casuale oppure indicando una lista di siti in cui devono essere presenti tutte le repliche master di ogni lfn. Inoltre possiamo anche specificare se riempire tutti gli SE's, in maniera casuale, con repliche dei vari file. Altri parametri riguardano la grandezza della coda dei job dei CE's e il tempo che impiegano i CE's per elaborare un file. Ve ne sono poi alcuni che caratterizzano il funzionamento dell'economic model. Infine possiamo specificare le modalità di visualizzazione dei risultati così come l'uso della GUI e della sua mappa, specificata nel quarto file di configurazione, che vedremo meglio più avanti. Al solito si rimanda alla guida del programma [18] per una più accurata descrizione di tutti i parametri. Di seguito sono riportati alcuni esempi dei file di configurazione.

File di configurazione principale

```
 #
# This file contains all the parameters required for OptorSim
# Using the Properties class to store this information every space
# in the key must be preceeded by a backslash.
#
grid\ configuration\ file = examples/gianni_grid.conf
job\ configuration\ file = examples/gianni_job.conf
number\ of\ jobs = 20
#
# The choice of the scheduling alorithms for the RB is:
# (1) simple
# (2) getAccessCost() + job queue (RB calls getAccessCost multiple times)
# (3) getAccessCost() + job queue (only one call to getAccessCost)
scheduler = 1
#
# The choice of optimisers is:
# (1) SimpleOptimiser - no replication.
# (2) AlwaysReplicateOptimiser - always replicates, deleting oldest file.
# (3) DeleteLeastAccessedFileOptimiser - always replicates, deleting least
#        accessed file.
# (4) EcoModelOptimiser - replicates when eco-model says yes, deleting
#        least valuable file.
```

```
# (5) EcoModelOptimizer Zipf-like distribution
optimiser = 2
#
# automatically multiplied by scale factor
#
dt = 10000000
#
# 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 = 8,5,3
#
# 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 = 1000
#
# 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 = 1
#
# 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 = 1000
#
##################
# Auction stuff #
##################
#
auction\ flag = yes
# actually the number of sites contacted
hop\ count = 50
timeout = 500
timeout\ reduction\ factor = 0.4
nested\ auction\ chance = 1.0
pin\ time\ limit = 1000
#
# Outputs auction information to auction.log. Can slow the
# simulation down a little bit.
#
auction\ log = yes
#
########################
# Histogramming stuff #
########################
#
# These are all multiplied by scale factor
```

```
#
no\ of\ jobs\ histo\ upper\ limit = 250000
job\ time\ vs\ time\ histo\ upper\ limit = 20000000
all\ jobs\ histo\ upper\ limit = 3000000
#
#############
# 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
#
# end
#
```

File di configurazione dei job

```
# Simple jobs for the 2 CEs in the simple network
# that work on one single logical file
# It’s a modified version of simple_job.conf
# File Table
#
\begin{filetable}
File1 1000 1
File2 1000 1
File3 1000 1
File4 1000 1
File5 1000 1
\end
#
# Job Table
# A job name and a list of files needed.
#
\begin{jobtable}
job1 File1 File2 File3 File4 File5
```

```
job2 File1 File2 File3 File4 File5
job3 File1 File2 File3 File4 File5
job4 File1 File2 File3 File4 File5
job5 File1 File2 File3 File4 File5
job6 File1 File2 File3 File4 File5
job7 File1 File2 File3 File4 File5
job8 File1 File2 File3 File4 File5
job9 File1 File2 File3 File4 File5
job10 File1 File2 File3 File4 File5
\end
#
# CE Schedule Table
# CE site id, jobs it will run
#
\begin{cescheduletable}
2 job1 job3 job5 job7 job9
7 job2 job4 job6 job8 job10
\end
#
# The probability each job runs
#
\begin{jobselectionprobability}
job1 0.1
job2 0.2
job3 0.3
job4 0.4
job5 0.5
job6 0.6
job7 0.7
job8 0.8
job9 0.9
job10 1.0
\end{jobselectionprobability}
```

File di configurazione della griglia

```
# 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
#
# I’ve modified site 1 respect to the original simple_grid.conf#
 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 di mappa

```
 # This file contains info on the map frame
#
# The file with the map picture
examples/gui/europe.jpg
# The width of the map
663
# The height of the map
612
# Site number, x and y coords, site(s) or router(r), name
0 216 172 s Manchester
1 257 230 r
2 386 134 r
3 322 213 s NIKHEF
4 317 254 r
5 397 234 r
6 264 314 r
7 281 404 s Lyon
8 329 352 s CERN
9 378 342 r
10 398 402 r
11 419 378 s Padova
12 420 412 s Bologna
13 476 561 s Catania
14 353 397 s Torino
```

```
15 377 379 s Milano
16 230 211 s RAL
17 437 75 s NorduGrid
```

6.3.2 Uso della GUI

Abbiamo visto che tramite il file di configurazione dei parametri è possibile attivare l'uso della GUI. A tal fine, sempre nello stesso file di configurazione, è presente l'indicazione di un quarto file di configurazione, quello contenente le informazioni sulla mappa da usare nella GUI.

Negli esempi forniti con OptorSim vi sono due mappe: una riguardante l'EDG testbed 1, e l'altra invece relativa al progetto inglese GridPP [10]. Nel file parameters.conf appena visto è indicata la prima mappa, contenuta nel file /examples/gui/europe.coords. Un map file deve contenere prima di tutto informazioni riguardo l'immagine della mappa, ovvero il file che la contiene e le sue dimensioni. Il resto è costituito da informazioni che riguardano i siti: le loro coordinate sulla mappa, il tipo ('s' per Grid site, un sito con un CE e/o un SE, 'r' per network router) e l'eventuale nome.

6.3.3 Simulazione ed analisi dei risultati

All'avvio OptorSim mostra subito un Pannello di Controllo (figura 6.5) dal quale si può dar inizio alla simulazione, metterla in pausa e terminarla.

Durante la simulazione, non appena il Resource Broker assegna un job, nel pannello di controllo viene indicato il sito che lo riceve. Il progresso della simulazione vera e propria può essere seguito nella finestra di monitoraggio della rete (figura 6.6). In questa vi è la mappa definita precedentemente, con l'indicazione dei vari siti e dei link che li collegano, la cui banda era stata indicata nel file di configurazione della griglia. I link hanno un diverso spessore che ne riflette la banda specificata. Inizialmente i link sono tutti di colore blu. Durante la simulazione invece il colore dei link cambia in modo tale da riflettere la situazione di traffico generato nella rete dall'esecuzione dei job, variando dal blu (traffico leggero) al rosso (traffico pesante).

Infatti quando un job viene eseguito su un certo CE può richiedere la creazione di nuove repliche o accessi remoti. Tutto ciò contribuisce ad aumentare il traffico di dati nei vari link della rete. La scelta dell'algoritmo di replicazione può avere un notevole impatto sul traffico generato; un buon algoritmo dovrebbe quindi mantenere i link più vicino possibile al colore blu durante tutto l'arco della simulazione.

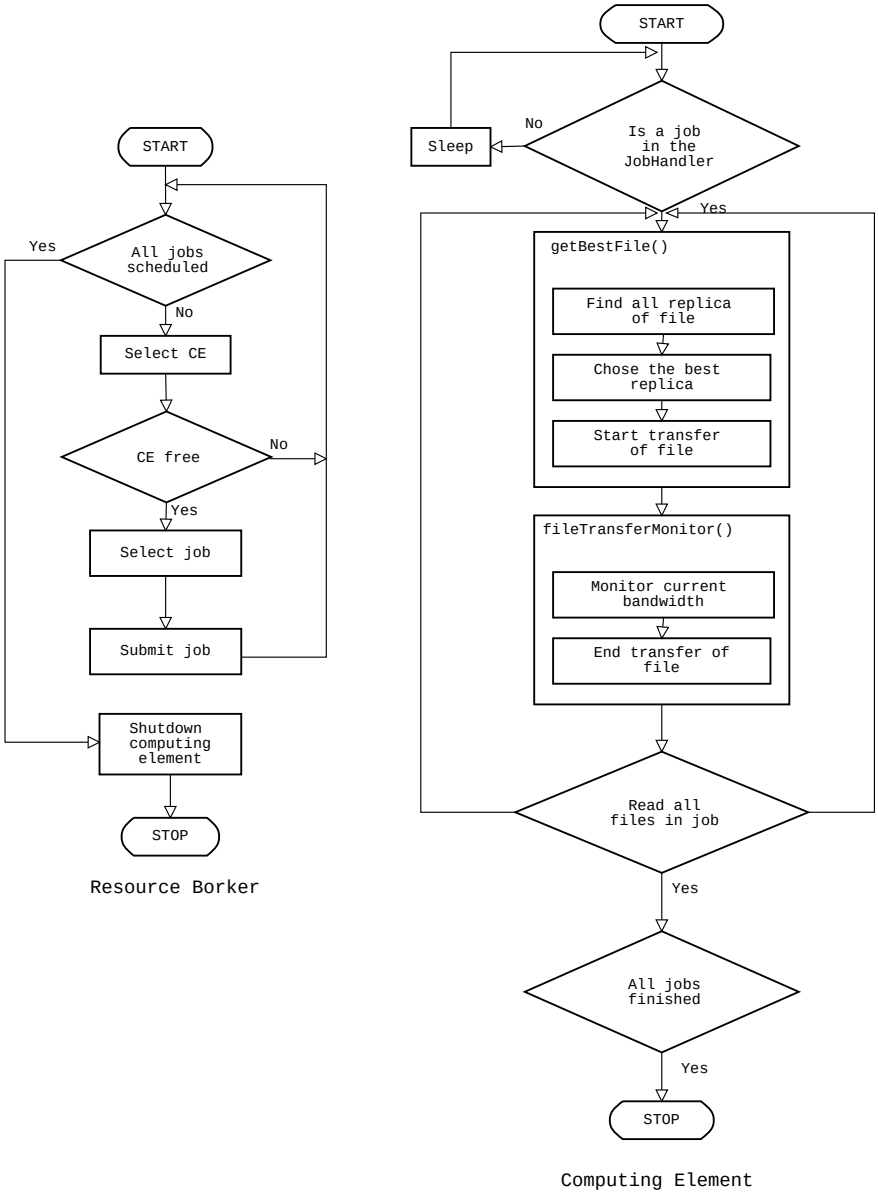


Figura 6.4: Diagrammi di flusso del Resource Broker e del thread di un Computing Element

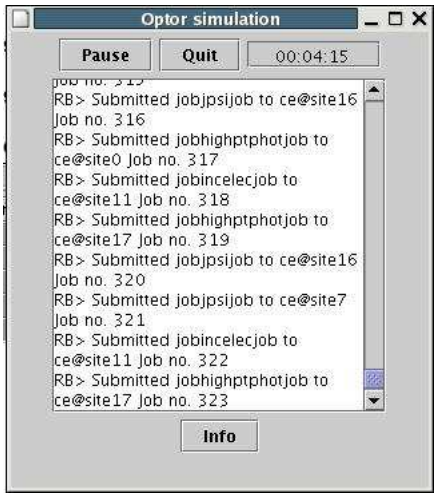


Figura 6.5: Pannello di Controllo

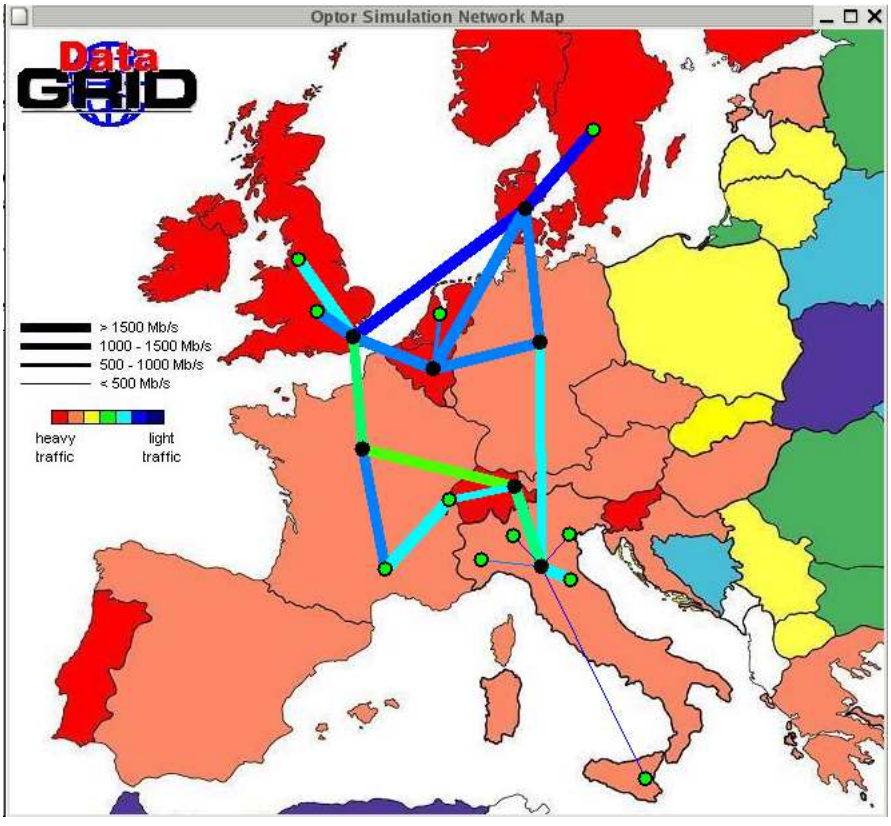


Figura 6.6: OptorSim Network Monitoring Window

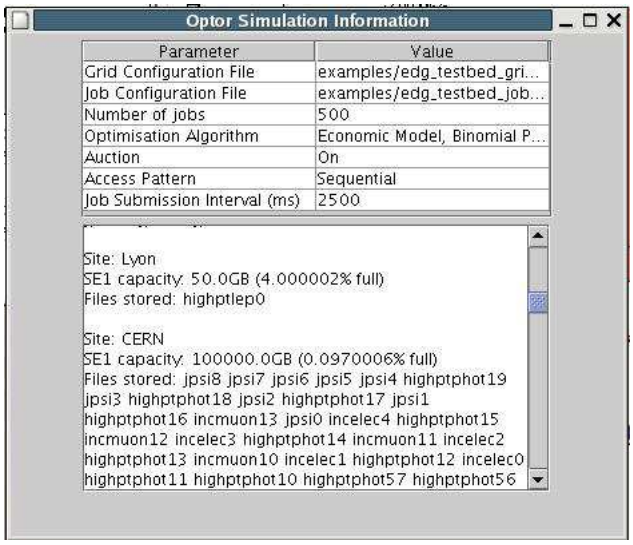


Figura 6.7: OptorSim Simulation Information Window

Dal pannello di controllo si può accedere, tramite il pulsante Information, alla OptorSim Simulation Information Window (figura 6.7) che visualizza informazioni in tempo reale sullo stato di ogni sito; cliccando su un sito indicato nella mappa, infatti, verranno mostrati i file memorizzati sul proprio SE e la percentuale di carico di quest'ultimo.

Alla fine della simulazione, nella directory *results*, verranno creati degli istogrammi che riassumono i risultati della simulazione appena eseguita. Questi possono anche essere visti durante la simulazione (figura 6.8, settando il parametro *histogram browser* nel file di configurazione principale.

Gli istogrammi sono di tre tipi e vengono raccolti in tre diverse directory:

- nojobs vs jobtime: per ogni tipo di job in ogni CE, ci indica come varia il job time, ovvero il tempo necessario per eseguire il job. Una certa quantità di job, quelli che accedono solamente repliche locali, saranno eseguiti con tempo nullo. Per altri invece il tempo varia in relazione al traffico di rete

- jobtime vs time: riassume l'andamento del tempo di esecuzione di un dato tipo di job ad uno specifico CE durante il corso della simulazione. Con l'uso di tecniche di ottimizzazione ci dovremmo aspettare un andamento decrescente ovvero, durante il corso della simulazione, il job time dovrebbe diminuire man mano che le repliche vengono distribuite in maniera ottimale

- alljobs: contiene l'istogramma del job time per ogni job eseguito.

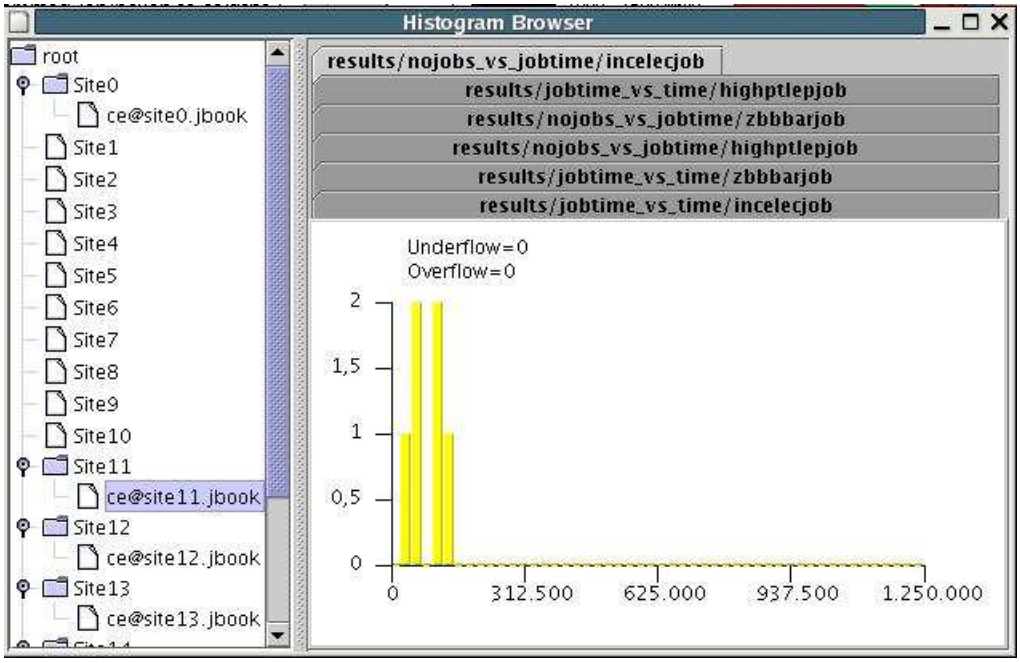


Figura 6.8: Histogram Browser Window

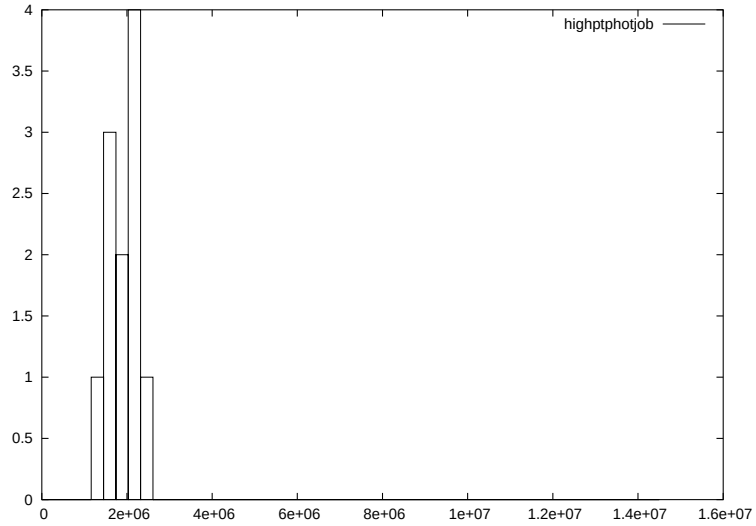


Figura 6.9: Istogramma nojobs vs jobtime per un dato tipo di job in un dato CE. Sulle ordinate compare il numero di istanze di tale job mentre sulle ascisse il tempo di esecuzione. Possiamo stabilire che, per quanto riguarda le varie istanze del job *highptphotjob* svolte in questo preciso CE, 4 istanze vengono eseguite con un tempo di circa 2e+06 ms, 3 istanze con un tempo leggermente inferiore e cosi via. Possiamo notare che non capita mai, a questo job su questo CE, di accedere repliche locali; lo possiamo capire dal fatto che non ci sono spike con valore nullo delle ascisse.

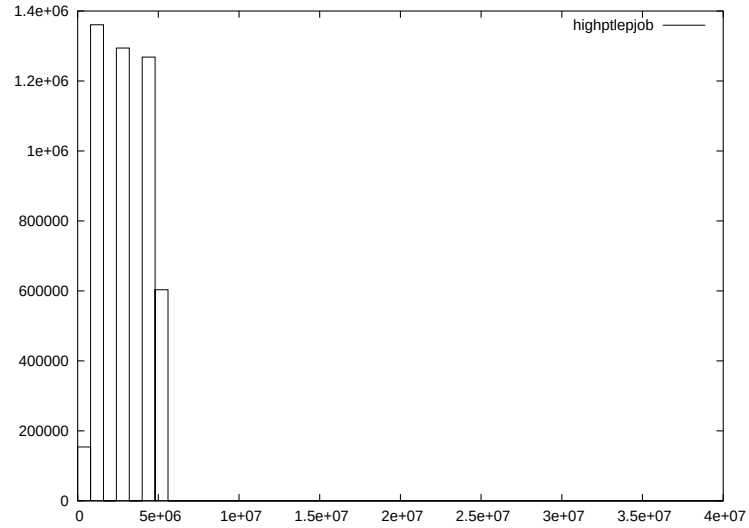


Figura 6.10: Istogramma jobtime vs time per un dato tipo di job (highptlepjob) in un dato CE.

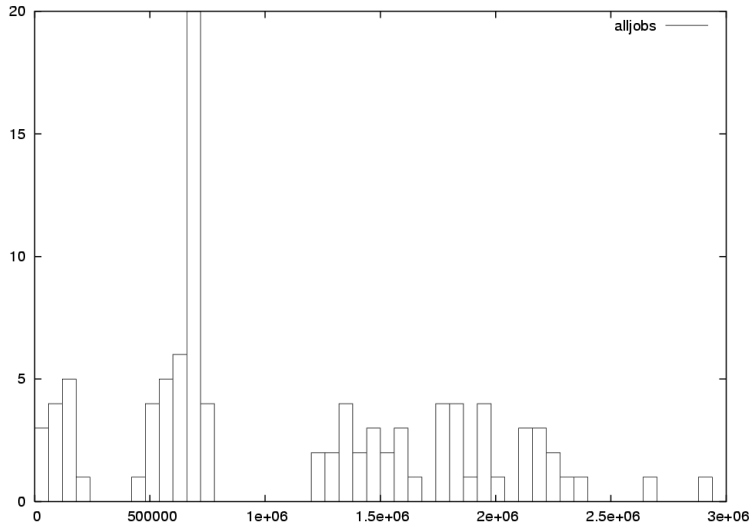


Figura 6.11: Istogramma dei tempi di job di tutti i job eseguiti durante la simulazione. Al solito, le ordinate contengono il numero di istanze (questa volta però non relativo ad un solo tipo di job ma a tutti) mentre sulle ascisse c'è il tempo di esecuzione.

Capitolo 7

Modelli di Consistency Service

In questo capitolo vengono illustrati i passi svolti nella creazione di due semplici modelli di Consistency Service da integrare in OptorSim. Questo dovrebbe costituire un punto di partenza per la simulazioni di modelli più evoluti che saranno la base di prova per un servizio da implementare in una Griglia reale. Il primo modello è un modello Eager (vedi sezione 4.2) anche se presenta una importante modifica che lo allontana un po' dalla definizione di Eager Replication ma che ci ha permesso di adattarlo all'ambiente di una Griglia. Il secondo modello è un tipico modello Lazy single-master (vedi sezione 4.3).

7.1 Progetto del Servizio di Consistenza

Le sezioni che seguono sono tratte da [21] e riguardano la progettazione di un servizio reale; dato che abbiamo cercato di tenere quanto più vicino possibile lo sviluppo dei moduli di simulazione al progetto del servizio reale, queste informazioni sono utili anche ai fini della simulazione e costituiscono un buon modo per riprendere i concetti già espressi nei capitoli precedenti. In particolare rivedremo con attenzione il problema dell'identificazione delle repliche.

Nel seguito faremo riferimento ad un generico ambiente Grid, dove abbiamo tre tipi di nodi: Computing Element (CE), Storage Element (SE) e User Interface (UI) (vedi sezione 3.2). I servizi che compongono il Middleware Grid forniscono alcune importanti funzionalità come il job-scheduling, l'allocazione delle risorse e la gestione dei dati. L'ultima classe è quella che comprende il Replica Management Service (RMS) e il Consistency Service (CS o RCS).

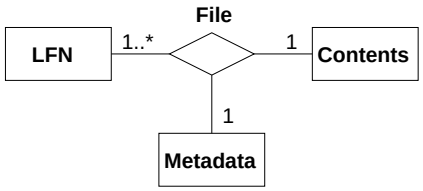


Figura 7.1: Una visione astratta di un file

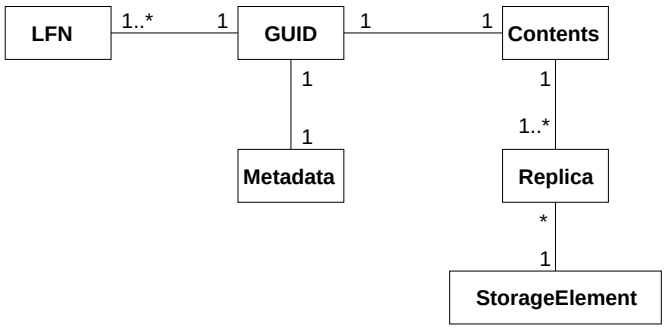


Figura 7.2: File e repliche

7.1.1 Il Servizio di Consistenza in una Griglia

Abbiamo già visto in dettaglio come è strutturato e come lavora il RMS (Reptor, sezione 5.4.1). Il concetto di "file" in una Griglia può essere formalizzato come una associazione tra il suo contenuto, con uno o più logical file name (lfn) e con i suoi metadati (attributi del file). Questa associazione è riassunta in figura 7.1.

La replicazione implica l'esistenza di più copie del contenuto del file presso diversi Storage Element. L'identificatore unico (GUID vedi sezione 5.2.2) ci permette di identificare l'insieme di repliche di un certo file e i vari logical file name che possono essere usati per far riferimento al file, come illustrato in figura 7.2.

Possiamo ridurre il modello a quello di figura 7.3, dove il contenuto di un file diventa attributo di una replica, insieme al suo nome. L'attributo *name* di una replica è il nome che identifica in maniera univoca la replica, che a seconda della convenzione sui nomi utilizzata avevamo chiamato physical file name o semplicemente file name (vedi sezione 5.2.1).

I diversi tipi di nomi corrispondono a diversi livelli di astrazione: il logical file name identifica il file come entità logica ed è il nome (che può variare) con cui il file è conosciuto dalle applicazioni a dagli utenti finali, mentre il GUID è usato dal Middleware Grid e il replica name localizza l'istanza fisica di una replica del file.

L'associazione univoca tra un GUID e il contenuto di una replica implica che tutte le

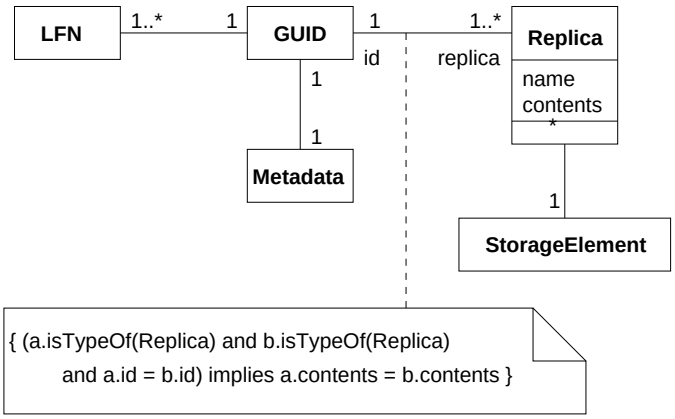


Figura 7.3: File e repliche

repliche con lo stesso GUID (cioè repliche dello stesso file) devono avere lo stesso contenuto. È compito del Servizio di Consistenza assicurare che questo vincolo sia soddisfatto, nella maniera opportuna. La consistenza quindi, in maniera semplificata, può essere vista come una relazione che vale in un insieme di repliche se e solo se tutte le repliche hanno identico contenuto. Quando una replica viene aggiornata (modificata), la modifica deve essere applicata a tutte le altre repliche. Chiameremo *file update* il processo di modifica di una replica e *update propagation* il processo di applicazione della modifica a tutto l'insieme di repliche. Abbiamo già visto nel capitolo 4 le varie soluzioni che possiamo adottare adottare per implementare meccanismi di file update e update propagation.

7.1.2 Principi di progetto

Meccanismi che assicurano la consistenza dei dati esistono in diversi DBMS distribuiti (li abbiamo visti in dettaglio nella sezione 4.5), ma per quanto ne sappiamo non è ancora disponibile un tale servizio per una Griglia. Durante la progettazione di tale servizio abbiamo seguito le seguenti linee guida:

- fornire una interfaccia di programmazione ad alto livello per semplificare l'utilizzo , nascondendo i dettagli implementativi
- fornire diversi modelli di consistenza, adattabili alle esigenze di diversi utenti e VO
- integrare il servizio quanto più possibile con gli altri servizi Grid, primo fra tutti il Replica Manager
- integrare il servizio con i diversi storage system utilizzati in maniera da interferire al minimo con le soluzioni esistenti

- realizzare un servizio quanto più possibile distribuito, ed usare il controllo centralizzato in maniera limitata per aumentare la tolleranza ai guasti e diminuire i colli di bottiglia.

7.1.3 Utilizzo del Servizio ed Architettura di Base

Partiremo dalla prospettiva di un cliente del servizio per capire le funzionalità richieste e poi andremo nei dettagli della realizzazione. In generale, sono richieste le seguenti funzionalità:

- Interfaccia di Programmazione: fornisce i metodi principali che possono essere invocati dai clienti del servizio, che sono altri servizi Grid ma anche applicazioni utente
- Meccanismo di File Update: è il meccanismo incaricato di applicare le modifiche ad una singola replica
- Protocollo di Update Propagation: serve per propagare l'aggiornamento ai siti remoti che ospitano le altre repliche.

7.1.3.1 Utilizzo del servizio

In maniera coerente con i principi di progetto l'interfaccia di programmazione fornita dal servizio deve essere il più semplice possibile. Le applicazioni possono leggere e scrivere dei file attraverso una interfaccia di I/O, cioè un insieme di operazioni che dipendono da come il file è memorizzato(plain Posix file system piuttosto che un file system distribuito come NFS o AFS). Queste operazioni non coinvolgono il nostro Servizio di Consistenza.

Per quanto riguarda le operazioni eseguite su file replicati, il coinvolgimento del Servizio di Consistenza è necessario. Le operazioni di lettura su tali file possono essere diverse dalle convenzionali operazioni di lettura su file: dato che una operazione di lettura può essere fatta su ogni replica, un utente potrebbe voler controllare lo stato di tale replica, in particolare se il suo contenuto è aggiornato. In generale, un possibile modo di implementare una tale procedura potrebbe essere quello di far precedere la lettura da una operazione del tipo:
consistencyStatus = getStatus(replicaName)
Il Servizio di Consistenza implementa la funzione *getStatus()* effettuando una query ad un catalogo interno che può restituire i valori *stale* o *up-to-date*.

In generale, il coinvolgimento del Servizio di Consistenza nelle operazioni di lettura è fortemente legato al modello usato: alcuni modelli Lazy possono prevedere di effettuare le letture senza utilizzare il servizio, altri possono implementare delle politiche più complesse che coinvolgono le versioni dei file.

Una operazione di scrittura su un file replicato la indicheremo come una operazione di *update*. In questo caso, il cliente che vuole aggiornare un file deve interagire con il Servizio di Consistenza, seguendo lo schema seguente:

1. ottenere il permesso di modificare un file logico
2. ottenere una copia locale di una replica su cui lavorare
3. modificare la copia locale tramite l'interfaccia di I/O
4. invocare i metodi del Servizio di Consistenza per aggiornare il file logico, applicando il meccanismo di file update alla replica selezionata e successivamente il meccanismo di propagazione dell'aggiornamento.

7.1.3.2 Meccanismo di File Update

Per prevenire anomalie negli accessi ai file (letture e scritture) è necessario utilizzare un meccanismo di locking. Per implementare il meccanismo di File Update abbiamo scelto di utilizzare un metodo content-transfer (vedi sezione 4.3.1.2), in particolare un total file replacement, che è più semplice e pensiamo che meglio si adatti ad un ambiente eterogeneo come la Griglia.

7.1.3.3 Update Propagation Protocol

Ci sono diverse possibilità per realizzare un protocollo di propagazione degli aggiornamenti, e li abbiamo visti in dettaglio nelle sezioni 4.2 e 4.3. Successivamente vedremo quali sono state le scelte per i modelli da simulare.

7.1.3.4 Architettura del Servizio

Basandosi su ciò che è stato detto nelle precedenti sezioni, possiamo passare ad illustrare l'architettura di base del servizio (figura 7.4) ed i componenti impiegati:

- Consistency Service: è il servizio che fornisce il punto di ingresso principale per l'applicazione utente attraverso il Consistency Service Client. Mette a disposizione i metodi principali per aggiornare i file, metodi che saranno poi implementati con l'ausilio degli altri componenti dell'architettura
- Local Consistency Service: situato accanto ad ogni SE, permette di aggiornare localmente un file e di partecipare al protocollo di propagazione degli aggiornamenti

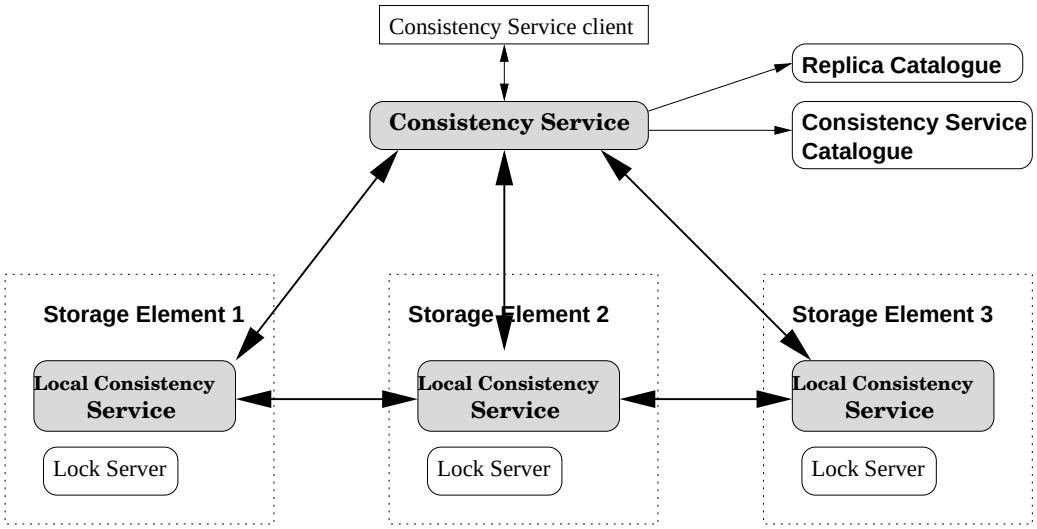


Figura 7.4: Architettura di base del Servizio di Consistenza

- Lock Server: implementa il servizio di locking richiesto per serializzare gli accessi ai file. È responsabile di gestire le richieste di lock/unlock sui file memorizzati presso lo SE
- Replica Catalog: è un servizio esterno con cui il Consistency Service deve collaborare per ricavare le informazioni sulla localizzazione delle repliche
- Consistency Service Catalog: è il catalogo interno al servizio che si occua di gestire tutte le informazioni necessarie all'implementazione dei meccanismi di file update e update propagation.

7.2 Implementazione dei Modelli da Simulare

Seguendo i principi di progetto appena espressi, e basandosi sull'architettura vista, abbiamo implementato un modello di Servizio di Consistenza che può utilizzare due protocolli, uno sincrono e uno asincrono. In ogni caso, il punto di vista dell'applicazione utente che utilizza il servizio non cambia. Nella figura 7.5 è illustrato un diagramma che riassume i passi svolti, durante la simulazione, dal thread che simula il funzionamento di un job. In figura 7.6 riportiamo il diagramma delle classi utilizzato per l'implementazione dei modelli da simulare.

Ulteriori dettagli sui metodi implementati compresi diagrammi di sequenza che mostrano alcuni scenari tipici di utilizzo, sono inclusi nell'appendice A, che riporta il testo del

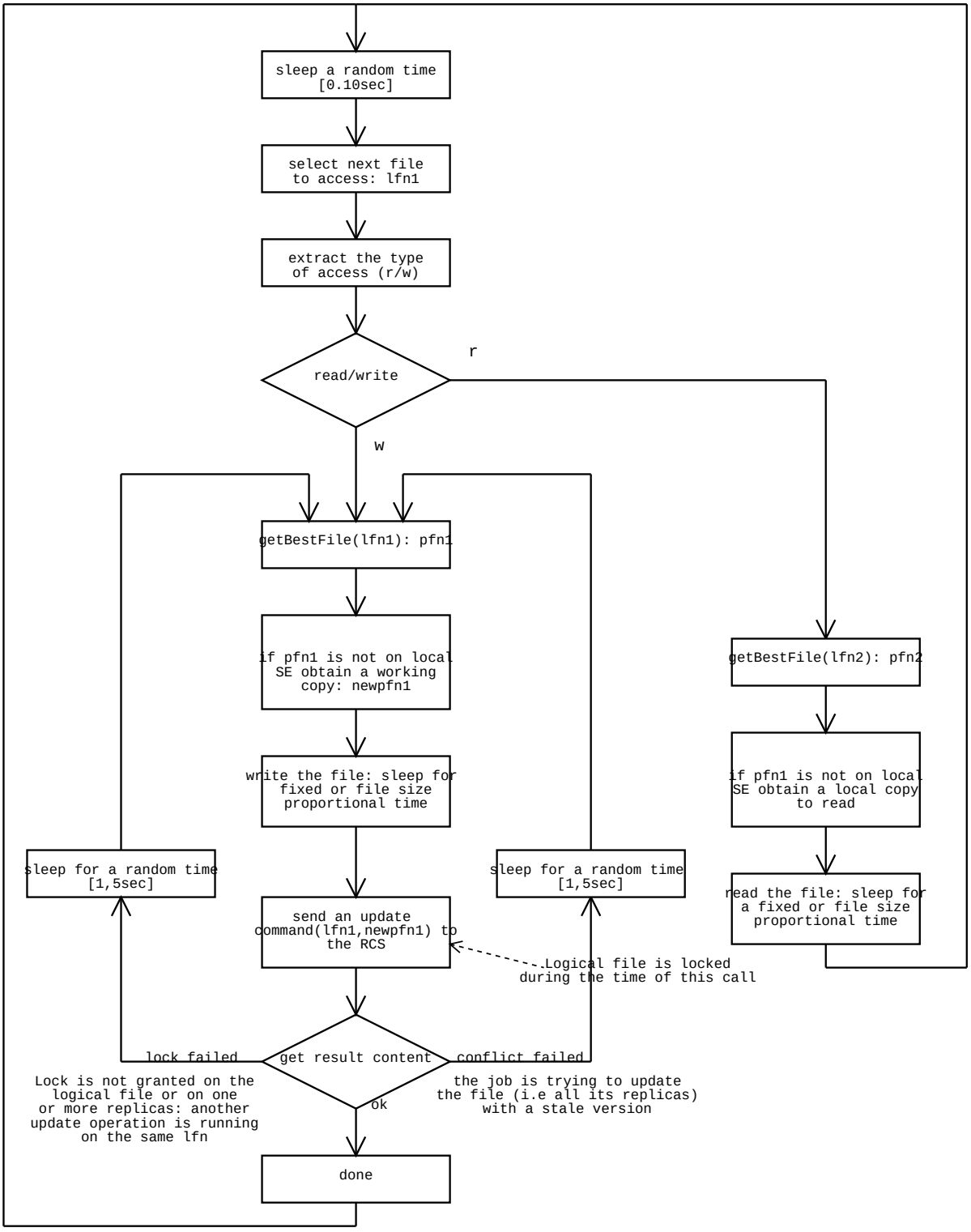


Figura 7.5: Diagramma di flusso di un generico job che utilizza il Servizio di Consistenza

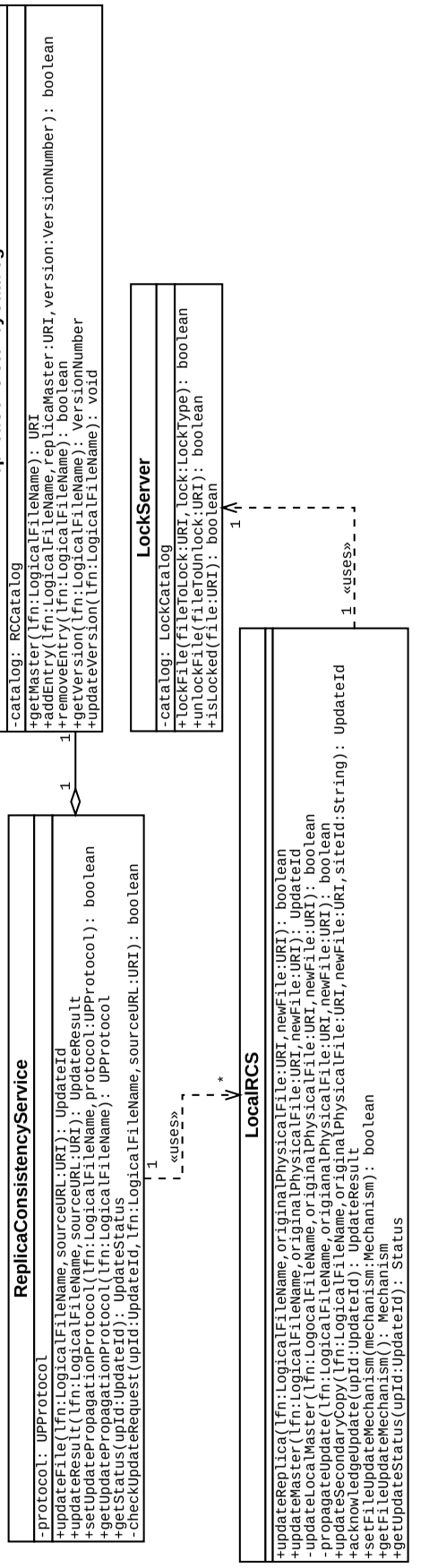


Figura 7.6: Diagramma delle classi del Servizio di Consistenza

documento [34]. Nelle sezioni che seguono cercheremo di dare un idea del funzionamento di base dei due modelli.

7.2.1 Modello Sincrono

Il primo modello, che abbiamo chiamato impropriamente sincrono, realizza un protocollo di consistenza simile ai tipici protocolli sincroni, ma con una importante differenza: l'applicazione utente (o job) prima di modificare un file deve recupare una copia di una certa replica, e lavorare su di essa localmente [1]. Una volta eseguita la modifica l'applicazione dovrà invocare il metodo di file update messo a disposizione del Consistency Service per aggiornare la replica. In questo caso, la chiamata ritorna all'utente solo nel momento in cui il Consistency Service si è incaricato di aggiornare la replica selezionata e tutte le altre, con l'ausilio dei Local Consistency Service.

Durante l'aggiornamento del file e la propagazione degli update, il Replica Consistency Service pone un lock sul file logico, impedendo ogni altra operazione su qualsiasi altra replica dello stesso file che potrebbe generare un conflitto. La gestione dei lock sui file logici viene realizzata dal Replica Consistency Service tramite il Consistency Service Catalog. In questo modo siamo in grado di assicurare la corretta applicazione degli aggiornamenti evitando delle anomalie di serializzazione.

Abbiamo detto però che questo modello non è un vero modello Eager (sincrono). Infatti, un utente che sta modificando una replica potrebbe invocare il metodo per aggiornarla quando la stessa cosa è stata fatta precedentemente da un altro utente per lo stesso file: in questo caso il Replica Consistency Service, per mantenere il corretto ordine di applicazione degli aggiornamenti deve necessariamente usare un catalogo in cui memorizzare le versioni correnti dei file logici. In questo modo, ad ogni corretta applicazione di un aggiornamento a un file, il suo numero di versione aumenta. Il numero di versione è associato al file logico e ad ogni replica viene associato il numero di versione corrente al momento della sua creazione. Se il Consistency Service riceve una richiesta di aggiornamento che può causare un conflitto (perché un utente tenta di aggiornare una versione che è già stata aggiornata da un altro utente) la richiesta viene negata, e il metodo ritorna con una indicazione di fallimento. A questo punto l'utente, per portare a termine il proprio lavoro, dovrà recuperare nuovamente la versione aggiornata del file, riapplicare le sue modifiche, e chiamare nuovamente il metodo di aggiornamento del Consistency Service. Questo protocollo viene esemplificato in figura A.1 nell'appendice A.

Può inoltre succedere che una operazione di lettura trovi dei dati non aggiornati, perché è stata selezionata una replica di un file logico per il quale una operazione di aggiornamento

[1] Questa copia non è una nuova replica ma soltanto una copia di lavoro che non viene quindi registrata nel Replica Catalog

è in corso, ma non ha ancora aggiornato la replica selezionata per la lettura. In questo caso si ha quella che viene chiamata una lettura “stale”, ovvero eseguita su un dato non ancora aggiornato.

Il numero di conflitti e il numero di letture stale che avvengono dipendono dal contesto in cui viene usato il Servizio di Consistenza e possono determinarne la qualità; in particolare la frequenza degli accessi ai file (letture e scritture) e il numero di repliche usate sono i parametri che hanno più influenza.

7.2.2 Modello Asincrono Single Master

Per implementare un modello Lazy abbiamo invece scelto l’opzione che prevede un singolo master: questo diminuisce la complessità del problema evitando la questione della risoluzione dei conflitti tra i vari master. In questo caso, il metodo di file update chiamato dall’applicazione utente ritorna non appena la replica master viene correttamente aggiornata; sarà poi compito del master LCS aggiornare le altre repliche. In figura A.2 in appendice A è mostrato il diagramma di sequenza di un tipico scenario.

Anche in questo caso possiamo avere a che fare con conflitti sugli aggiornamenti e con letture stale, ed essendo il protocollo asincrono ci possiamo aspettare un incremento di questi valori.

7.2.3 Risultati preliminari

In questa sezione riporteremo l’esito delle simulazioni fatte alla fine di questo lavoro. I dati che riportiamo sono frutto di simulazioni eseguite su piccola scala e sono quindi da considerarsi dei risultati preliminari, in attesa di ulteriori simulazioni eseguite su larga scala con strumenti hardware/software adeguati. Abbiamo effettuato diverse prove utilizzando il modello sincrono e variando la percentuale di operazioni di scrittura rispetto al numero totale di accessi ai file eseguiti dai vari job. In particolare abbiamo modificato i file di configurazione di OptorSim in modo da eseguire una prima serie di simulazioni con i seguenti parametria:

- griglia con 10 SE e 2 CE
- 10 diversi tipi di job
- 4 file logici
- protocollo di ottimizzazione “always replicate”
- 50 job eseguiti durante la simulazione

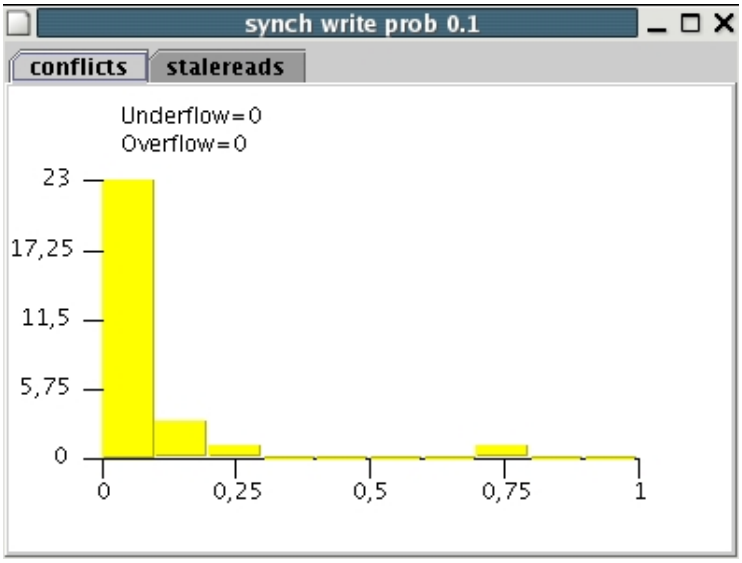


Figura 7.7: Istogramma del numero di conflitti con probabilità di scrittura 0.1

- probabilità di accesso in scrittura ai file variabile di simulazione in simulazione.

I messaggi prodotti dal simulatore sono stati modificati per poterli manipolare facilmente con script PERL con cui abbiamo ricavato le informazioni volute, in particolare riguardanti il numero di conflitti ed il numero di letture stale avvenute durante la simulazione. In figura 7.7, 7.8, 7.9 sono mostrati gli istogrammi relativi al numero di conflitti.

Gli istogrammi riportano sulle ordinate il numero di simulazioni e sulle ascisse il rapporto conflitti/numero di accessi ai file.

Possiamo vedere che la percentuale di conflitti rispetto al numero di accessi ai file è compresa tra 0 e 0,25, con valori occasionali intorno a 0.75. Possiamo anche notare, come era lecito aspettarsi, un incremento del numero dei conflitti all'aumentare del numero di accessi in scrittura.

Per quanto riguarda le letture stale i dati ottenuti ci hanno mostrato che solo in pochissimi casi (meno del 5% delle simulazioni effettuate) si ottengono letture stale con una percentuale che si aggira intorno al 2% degli accessi. A causa di questi bassi valori gli istogrammi che riguardavano le letture stale sono stati omessi; risultava infatti troppo piccolo il valore sulle ascisse tanto che l'istogramma mostrava praticamente un valore nullo per tutte le simulazioni.

In appendice B invece è riportata la descrizione di un piccolo progetto realizzato precedentemente a quello appena visto. Ci è servito per "fare pratica" con il codice del simulatore OptorSim e con gli strumenti software utilizzati per lo sviluppo.

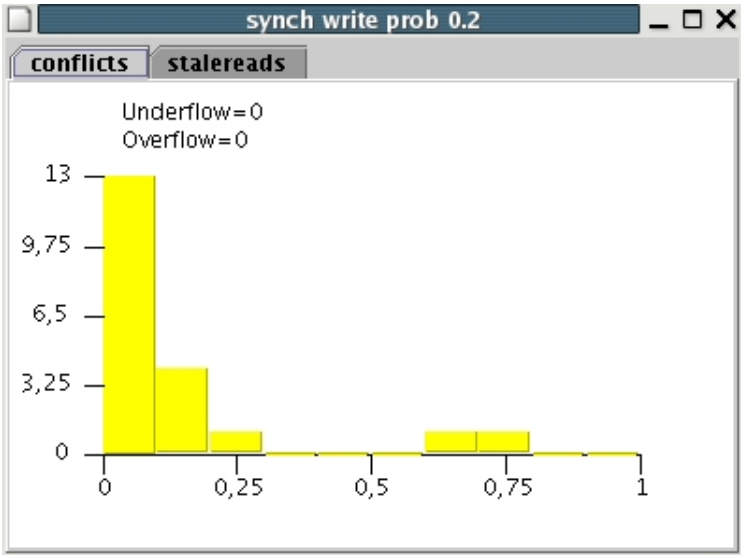


Figura 7.8: Istogramma del numero di conflitti con probabilità di scrittura 0.2

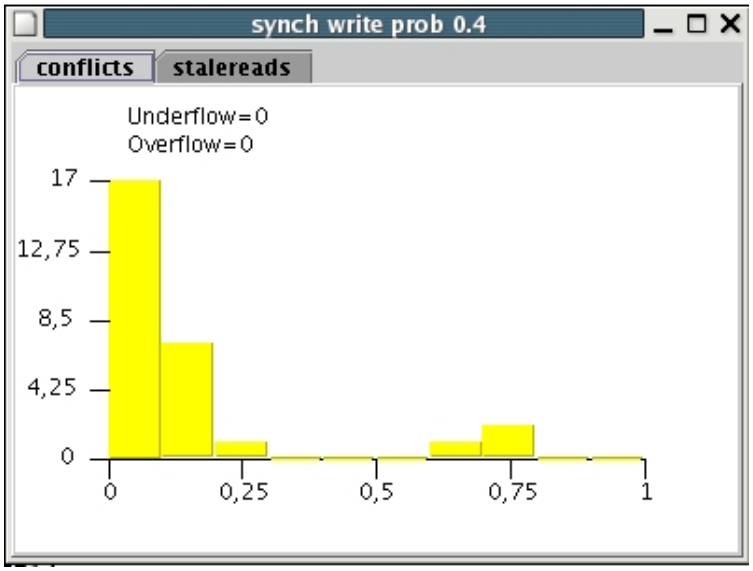


Figura 7.9: Istogramma del numero di conflitti con probabilità di scrittura 0.4

Capitolo 8

Conclusioni

Nel corso di questa tesi ho avuto modo di collaborare con la comunità dei ricercatori che si occupano di Grid Computing. Il Grid Computing è una disciplina abbastanza nuova che sta raccogliendo un notevole interesse da tutti gli operatori del settore dell'Information Technology; nel capitolo 2 abbiamo visto una rassegna dei più importanti progetti che sono in corso di svolgimento in tutto il mondo.

In particolare ho avuto la possibilità di lavorare a stretto contatto con alcuni ricercatori coinvolti nel progetto EU DataGrid, questo mi ha permesso di comprendere con più chiarezza i problemi legati allo sviluppo di un software di Grid Middleware.

Abbiamo studiato le problematiche connesse alla gestione dei dati in un ambiente replicato e di affrontare nel dettaglio il problema della consistenza delle repliche. Meccanismi di replica vengono oggigiorno utilizzati in molti sistemi ditribuiti, in particolare nei DBMS. Abbiamo analizzato le principali soluzioni fornite da alcuni dei più noti DBMS ed abbiamo realizzato che le differenze sostanziali che distinguono l'ambiente Grid da quello dei DBMS e degli altri sistemi distribuiti non ci consente di risolvere il problema della consistenza utilizzando le stesse tecniche. Abbiamo comunque fatto leva sulle conoscenze del problema della consistenza in quei settori per stabilire i requisiti di un Servizio di Consistenza in un ambiente Grid. Siamo passati poi alla realizzazione di due semplici modelli da simulare tramite il Grid Simulator OptorSim.

Il lavoro svolto in questa tesi è stato utilizzato per la pubblicazione di un articolo [20] al IX International Workshop on Advanced Computing and Analysis Techniques in Physics Research che si è tenuto presso l'High Energy Accelerator Research Organization (KEK) a Tsukuba (Giappone) nei primi giorni di Dicembre 2003, durante il quale ho avuto il piacere di presentare il lavoro svolto.

Sviluppi futuri

Per quanto riguarda le ulteriori possibilità di lavoro futuro riguardanti questa tesi, possiamo dire che ci sono ampi margini di sfruttamento. I modelli per il simulatore realizzati possono sicuramente essere migliorati ed ampliati. In particolare sarebbe interessante raccogliere un maggior numero di dati tramite simulazioni eseguite su cluster di computer, per poter meglio valutare l'impatto di un Servizio di Consistenza sulle prestazioni di una Griglia. Potrebbe essere interessante anche valutare la possibilità di realizzare modelli più complessi, che coinvolgono più master e che utilizzano tecniche di risoluzione dei conflitti.Anche l'impiego di meccanismi di voting dovrebbe essere valutato per gestire il caso dei nodi disconnessi.

Ringraziamenti

Concludo la tesi con i dovuti ringraziamenti a chi mi ha sostenuto durante questo lavoro. Un grazie particolare al mio relatore Andrea Domenici, che con serietà e cordialità ha fornito il continuo appoggio grazie al quale è stato possibile realizzare questo lavoro in modo sereno e piacevole. Insieme a lui devo ringraziare Flavia Donno, Heinz e Kurt Stockinger per i continui scambi di idee e per la fiducia mostrata nei miei confronti, specialmente in occasione della missione giapponese. Grazie anche per aver compreso e risposto a tutte le mail...

Per il supporto fornito durante tutta la non breve carriera universitaria il primo grazie va alla mia famiglia: grazie a Mamma per avermi dato la possibilità di realizzare tutto questo e per avermi sostenuto in tutti i modi possibili anche nei momenti più difficili; ad Ale e a Valentina, che mi hanno liberato la camera per lavorare meglio sulla tesi e che presto mi faranno diventare zio e a Benny, July e Schizzo. Grazie ai mitici Cri e SZ, per il continuo supporto morale e materiale (cioccolata inclusa). Grazie Lila, è stato un piacere telefonarti alla fine di ogni esame e ricevere i tuoi complimenti. Grazie anche a Gianni e Maria, Simonetta, Donatello, Michele, Serena e Tania.

Grazie ai miei amici: Alessandro, Chiara, Lisa, Massimo e Valerio, che hanno sopportato i continui -Oggi non posso, devo studiare-. Grazie alla lunga lista di amici e compagni di università per la compagnia e gli appunti forniti, perdonatemi se non vi elenco tutti ma siete davvero tanti e finirei col dimenticare qualcuno. Grazie a Marco e alla sua famiglia. Grazie anche a Stefano per la sua particolare amicizia, a Giulia e Renzo.

Dedico questo lavoro al mio Babbo.

Appendice A

Replica Consistency Service (RCS) - Interface v0.3

In questa appendice riportiamo il testo integrale del documento [34], che illustra l'interfaccia del servizio di Consistenza che dovrà essere sviluppato. La medesima interfaccia è stata utilizzata per implementare il modulo di consistenza da integrare in OptorSim.

Abstract

The following document provides details on the interfaces for the Replica Consistency Service. In detail, the basic user interface as well as the interfaces for each of the service components are given.

A.1 Introduction

The following document provides interfaces for following service components:

- Replica Consistency Service (RCS, main user interface)
- Local Replica Consistency Service (LRCS)
- Replica Consistency Catalogue (RCC)
- Lock Server

It is assumed that on each Storage Element one LRCS is located and thus the LRCS hostname can always be determined by the name of the physical file.

Blocking vs. Non-blocking Calls

For many of the methods stated in the following sections, there are non-blocking calls that return with a unique ID. It is then up to the caller to poll the state of the request since the method returns immediately. Example:

```
public updateID updateFile(LogicalFileName lfn,
                           URI sourceURL)
```

In order to avoid polling, there is also the possibility to use the equivalent blocking call that returns with the status of the request only when the request has been executed (successfully or unsuccessfully). Rather than returning an update ID, the method returns a detailed result about the request execution. Example:

```
public Result updateFile(LogicalFileName lfn,
                         URI sourceURL)
```

In the remainder of the document, only non-blocking calls are discussed and the blocking pendents are assumed.

A.2 Replica Consistency Service - User Interface

The following user interface is a very simplified one where all file updates are first done on a local copy (e.g. on a Worker Node on the Computing Element) and then the updated file is written back to the data storage, i.e. no direct updates on the Storage Elements are performed which seems to be a good scenario for a Grid usage. In addition, it has the advantage that file locking is very minimal and no specialised remote update tools are required for the Storage Element.

A.2.1 updateFile

Description

A file has already been modified (located at `sourceURL`) and now needs to replace (update) all existing replicas. The command returns an updateId that can be polled about the status of the update propagation progress (see Section A.2.2). In addition, the `LFN` for a given file needs to be provided. The RCS finds all existing physical files and updates

them in order to keep them identical to the file indicated by the location `sourceURL`. The file transfer to the is initiated (in case of asychnronous replication: to the master site).

Internal details: Based on the update propagation protocol (synch or asych), the RCS needs to call either `updateReplica` or `updateMaster`, respectively.

```
public updateId updateFile(LogicalFileName lfn,
                           URI sourceURL)
```

Parameters

- *lfn*: logical filename of the file to update. The file needs to be registered already in the Replica Catalogue.
- *sourceURL*: location of the file to be updated. It can either be a local or remote site.

Return value

- *updateId*: a unique ID assigned by the RCS that uniquely identifies the update request.

A.2.2 getStatus

Description

Get the status of the file update propagation for a given `updateId`.

```
public updateId getStatus(String updateId)
```

Parameters

- *updateId*: a unique ID that has been obtained from the RCS

Return Value

The return value can be one of the following (TODO):

- `inProgress`

- `masterUpdated`
- `done`
- `failed`

A.2.3 setUpdatePropagationProtocol

Description

Set the update propagation protocol.

```
public void setUpdatePropagationProtocol(Protocol protocol)
```

Parameters

- *Protocol*: the update propagation protocol can have one of the following values:
  - `synch`: synchronous update model
  - `asynchOneMaster`: asychnronous model with one master file
  - `asynchMultipleMaster`: asychnronous model with multiple master files

A.2.4 getUpdatePropagationProtocol

Description

Get the update propagation protocol used.

```
public Protocol getUpdatePropagationProtocol()
```

Return Value

- *Protocol*: the update propagation protocol can have one of the following values:
  - `synch`: synchronous update model
  - `asynchOneMaster`: asynchronous model with one master file
  - `asynchMultipleMaster`: asynchronous model with multiple master files

A.3 Local Replica Consistency Service

A.3.1 updateMaster

Description

Initiate an *asynchronous* update of the physical file (master) with the new file using the private method `updateLocalMaster`. The method first checks if the file already exists since the command `updateFile` is supposed to have transferred it already to the Storage Element managed by the LRCS. It is now the task of the LRCS to lock the master file, replace it with the new file and notify the secondary copies (using the command `propagateUpdate`). By keeping both files (new and old) there is still the possibility to roll back to the original version.

```
public updateId updateMaster(LogicalFileName lfn,
                             URI originalPhysicalFile,
                             URI newFile)
```

Parameters

- *lfn*: logical file name of the file to be updated
- *originalPhysicalFile*: location of the original file at the local Storage Element that needs to be updated.
- *newFile*: location of the new file that needs to replace the `originalPhysicalFile`.

Return Value

- *updateID*: The ID contains the status of the job.

A.3.2 updateReplica

Description

Initiate a *synchronous* update of the physical file. The method first checks if the file already exists since the command `updateFile` is supposed to have transferred it already to the Storage Element managed by the LRCS. It is now the task of the LRCS to lock the physical file, replace it with the new file and notify the RCS when once the update has

succeeded. By keeping both files (new and old) there is still the possibility to roll back to the original version.

```
public updateID updateReplica(LogicalFileName lfn,
                              URI originalPhysicalFile,
                              URI newFile)
```

Parameters

- *lfn*: logical file name of the file to be updated
- *originalPhysicalFile*: location of the original file at the local Storage Element that needs to be updated.
- *newFile*: location of the new file that needs to replace the `originalPhysicalFile`.

Return Value

- *updateID*: the updateID is unique for each LRCS and allows to uniquely identify the status of the request.

A.3.3 updateLocalMaster

Description

Based on the update propagation protocol update the local master. Note that this method is private and used only for *asynchronous* replication. In order to update the file, the Lock Server has to be contacted (`lockFile(URI, write)`), `unlockFile(URI, write)`.

```
private updateId updateLocalMaster(LogicalFileName lfn,
                                   URI originalPhysicalFile,
                                   URI newFile)
```

Return Value

- *updateID*: the updateID is unique for each LRCS and allows to uniquely identify the status of the request.

A.3.4 propagateUpdate

Description

Depending on the update propagation method (needs to be a configurable parameter for the service), the updates of the master are propagated to all secondary copies. Note that this method is private and can only be executed by the LRCS. For each of the replicas, the remote method `updateSecondaryCopy` has to be called.

```
private updateId propagateUpdate(LogicalFileName lfn,
                                 URI originalPhysicalFile,
                                 URI newFile)
```

Return Value

- *updateID*: the updateID is unique for each LRCS and allows to uniquely identify the status of the request.

A.3.5 updateSecondaryCopy

Description

Update the physical file (secondary copy) with the new file. This command can only be called by the master site. The command transfers the file and then updates it locally. In order to update the file, the Lock Server has to be contacted (`lockFile(URI, write)`), `unlockFile(URI, write)`. Only if a write lock is gained, the actual update process can take place.

```
public updateId updateSecondaryCopy(LogicalFileName lfn,
                                    URI originalPhysicalFile,
                                    URI newFile,
                                    String siteId)
```

Parameters

- *lfn*: logical file name of the file to update
- *originalPhysicalFile*: physical file at the LRCS to be updated

- *newFile*: new physical file at the site of the caller. This file is then transferred to the LRCS.
- *siteId*: the siteId is used in order to determine if the sender is the master site. The LRCS still needs to verify with the host and the Replica Consistency Catalogue the client of that command is really the master site.

Return Value

- *updateId*: Unique ID that identifies the transaction to the remote sites.

A.3.6 acknowledgeUpdate

Description

Once the update has been done (successfully or not) a notification message is sent to the requester of the update.

```
public UpdateResult acknowledgeUpdate(updateId)
```

Parameter

- *updateId*: unique ID to identify the request from a certain LRCS

Return Value

- *UpdateResult*: gives details about the update result.

A.3.7 setFileUpdateMechanism

Description

For each LRCS one can set the file update mechanism that determines in what way updates are propagated and how much inconsistency will arise. Possible file update mechanisms are:

- Simple file replacement

- Binary difference
- Log based Method (in a restricted way)

```
public boolean setFileUpdateMechanism(Mechanism mechanism)
```

Parameters

- *mechanism*: the file update mechanism can have one of the following values:
  - `simple`
  - `difference`
  - `log`

A.3.8 getFileUpdateMechanism

Retrieve the current file update mechanism.

```
public Mechanism setFileUpdateMechanism()
```

Return value

- *Mechanism*: the file update mechanism can have one of the following values:
  - `simple`
  - `difference`
  - `log`

A.3.9 getUpdateStatus

Description

Return the status of an update transaction.

```
public Status getUpdateStatus(updateId)
```

Parameter

- *updateId*: unique identifier for a certain update request

Return Value

Actual status of the transaction. Can have one of the following values:

- `inProgress`: local update is currently in progress
- `requestDenied`: local update is not possible due to several possible problems (e.g. dead lock avoidance)
- `done`: update has successfully finished
- `failed`: update has failed

A.3.10 setUpdatePropagationProtocol

The command entered from the user interface to the RCS (see Section A.2.3) needs to propagate to each of the LRCSs.

A.3.11 getUpdatePropagationProtocol

The command entered from the user interface to the RCS (see Section A.2.4) needs to propagate to each of the LRCSs.

A.4 Replica Consistency Catalogue

A.4.1 getMaster

```
public String getMaster(LogicalFileName lfn)
```

Parameter

- *lfn*: logical filename of the file to be updated

Return value

- *SURL*: Location of the physical file holding the master flag

A.4.2 setMaster

Description

Set the master flag for a given LFN/SURL pair.

```
public boolean setMaster(LogicalFileName lfn, URI SURL)
```

Parameters

- *lfn*: logical filename of the file to be updated
- *SURL*: physical location of the file to be assigned the master flag

A.5 Lock Server

A.5.1 lockFile

Description

Lock a file on a given Storage Element. The lock can either be a read or a write lock.

```
public boolean lockFile(URI SURL,
                        LockType lock)
```

Parameters

- *SURL*: physical file on the Storage Element that needs to be locked.
- *lock*: one can either specify a read or a write lock using one of the following types respectively: `readLock`, `writeLock`.

A.5.2 unlockFile

Description

Unlock a file on a given Storage Element. The lock can either be a read or a write lock.

```
public boolean unlockFile(URI SURL,
                          LockType lock)
```

Parameters

- *SURL*: physical file on the Storage Element that needs to be locked.
- *lock*: one can either specify a read or a write lock using one of the following types respectively: `readLock`, `writeLock`.

A.5.3 isLocked

Description

Return the lock status of a given file.

```
public boolean isLocked(URI SURL)
```

Parameter

- *SURL*: physical file on the Storage Element that needs to be locked.

Return Value

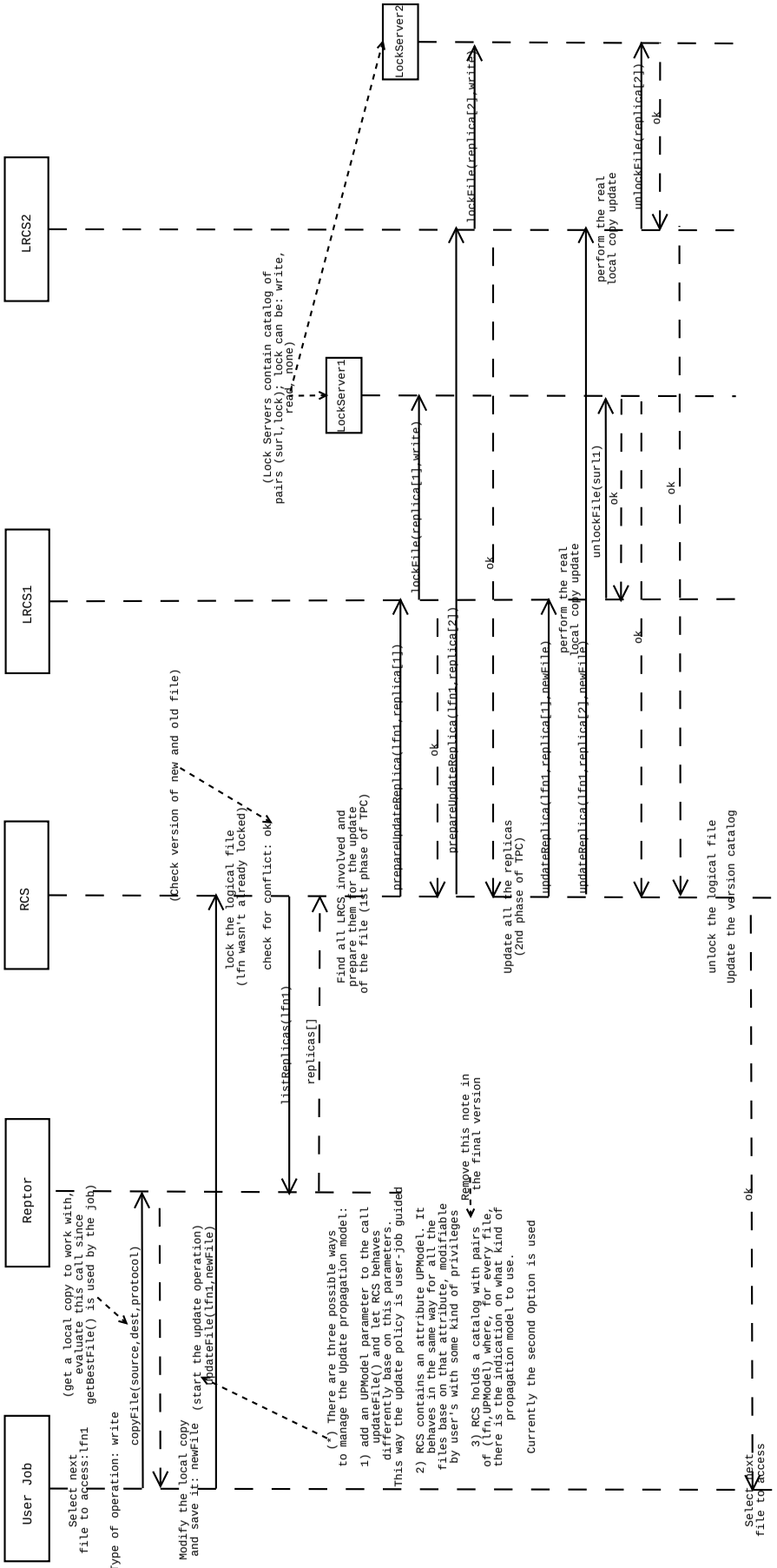
Returns `true` if the file is locked and `false` otherwise.

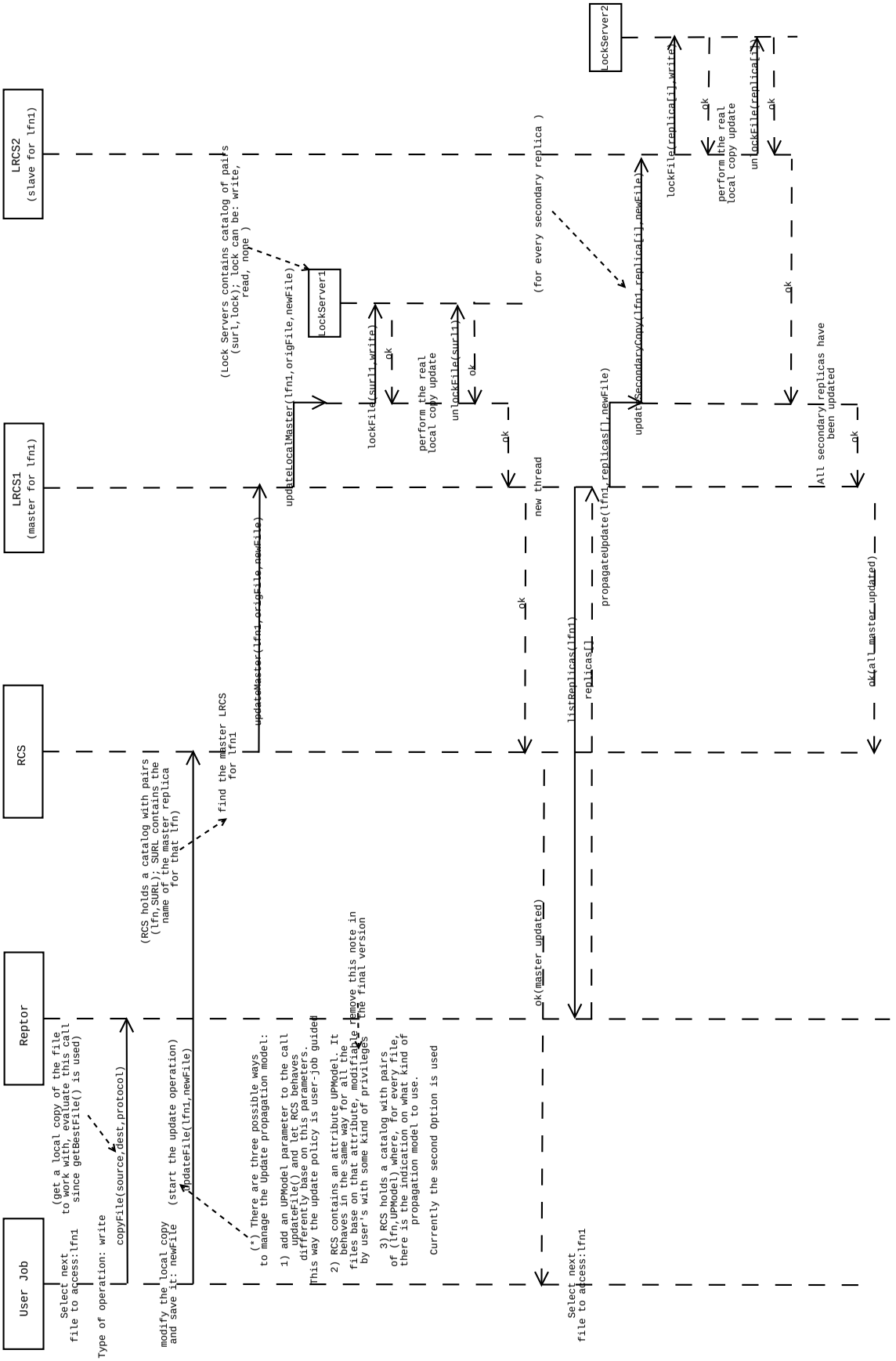
A.6 Command Sequence for Replication

In the following section, the sequence of the file updates is discussed for synchronous and asynchronous replication, respectively.

A.6.1 Synchronous Replication

A.6.2 Asynchronous Replication


User Job
Reptor
RCS
LRCS1
LRCS2
LockServer1
LockServer2
Select next file to access:lfn1
(get a local copy to work with, evaluate this call since getBestFile() is used by the job)
Type of operation: write
copyFile(source,dest,protocol)
Modify the local copy and save it: newFile
(start the update operation) updateFile(lfn1,newFile)
(check version of new and old file)
lock the logical file (lfn wasn't already locked)
check for conflict: ok
listReplicas(lfn1)
replicas[]
Find all LRCS involved and prepare them for the update of the file (1st phase of TPC)
(lock Servers contain catalog of pairs (surl,lock); lock can be: write, read, none)
prepareUpdateReplica(lfn1,replica[1])
lockFile(replica[1],write)
ok
prepareUpdateReplica(lfn1,replica[2])
lockFile(replica[2],write)
ok
Update all the replicas (2nd phase of TPC)
updateReplica(lfn1,replica[1],newFile)
perform the real local copy update
updateReplica(lfn1,replica[2],newFile)
perform the real local copy update
unlockFile(surl1)
ok
ok
unlockFile(replica[2])
ok
ok
unlock the logical file
Update the version catalog
ok
Select next file to access
(*) There are three possible ways to manage the Update propagation model:
1) add an UPModel parameter to the call updateFile() and let RCS behaves differently base on this parameters. This way the update policy is user-job guided
2) RCS contains an attribute UPModel. It behaves in the same way for all the files base on that attribute, modifiable by user's with some kind of privileges
3) RCS holds a catalog with pairs of (lfn,UPModel) where, for every file, there is the indication on what kind of propagation model to use.
Currently the second Option is used
Remove this note in the final version


User Job
Reptor
RCS
LRCS1
(master for lfn1)
LRCS2
(slave for lfn1)
LockServer1
LockServer2
Select next
file to access:lfn1
(get a local copy of the file
to work with, evaluate this call
since getBestFile() is used)
Type of operation: write
copyFile(source,dest,protocol)
modify the local copy
and save it: newFile
(start the update operation)
updateFile(lfn1,newFile)
(*) There are three possible ways
to manage the Update propagation model:
1) add an UPModel parameter to the call
updateFile() and let RCS behaves
differently base on this parameters.
This way the update policy is user-job guided
2) RCS contains an attribute UPModel. It
behaves in the same way for all the
files base on that attribute, modifiable
by user's with some kind of privileges
3) RCS holds a catalog with pairs
of (lfn,UPModel) where, for every file,
there is the indication on what kind of
propagation model to use.
Currently the second Option is used
(RCS holds a catalog with pairs
(lfn,SURL); SURL contains the
name of the master replica
for that lfn)
find the master LRCS
for lfn1
updateMaster(lfn1,origFile,newFile)
(Lock Servers contains catalog of pairs
(surl,lock); lock can be: write,
read, nope)
updateLocalMaster(lfn1,origFile,newFile)
lockFile(surl,write)
ok
perform the real
local copy update
unlockFile(surl)
ok
ok
new thread
ok(master updated)
listReplicas(lfn1)
replicas[]
propagateUpdate(lfn1,replicas[],newFile)
(for every secondary replica)
updateSecondaryCopy(lfn1,replica[i],newFile)
lockFile(replica[i],write)
ok
perform the real
local copy update
unlockFile(replica[i])
ok
ok
All secondary replicas have
been updated
ok
ok(all master updated)
Select next
file to access:lfn1

Appendice B

Modello di prova di RCS

In questa appendice riportiamo il primo modello che è stato implementato per prendere confidenza con il simulatore e gli strumenti software utilizzati per l'implementazione. Questo modello non rispecchia l'interfaccia proposta in appendice A e può considerarsi un piccolo progetto di prova. In questo semplice modello è implementato un protocollo di consistenza Eager (vedi sezione 4.2). Come abbiamo avuto modo di vedere i protocolli di Eager Replication assicurano un forte livello di consistenza aggiornando tutte le repliche di un dato file all'interno di una singola transazione atomica. In questo modo si evitano inconsistenze tra le repliche e la necessità di adottare degli algoritmi di riconciliazione per risolvere i conflitti. L'uso del locking in questo caso evita situazioni potenzialmente pericolose e le tramuta in attese o deadlock[1].

Semplici modelli di Eager Replication proibiscono l'aggiornamento di alcune repliche se almeno un nodo è disconnesso. Questa può rivelarsi una pessima scelta in un ambiente in cui i nodi cadono con facilità. Per ovviare a queste situazioni si usano i meccanismi di voting. Al momento, in OptorSim, si suppone che ogni nodo sia sempre attivo; questo introduce una notevole semplificazione rispetto ad una situazione reale. Un meccanismo per simulare la caduta di un nodo, in termini ancora da valutare, potrebbe essere implementato e questo introdurrebbe il problema dell'aggiornamento di un nodo nel momento in cui ritorna attivo[2].

In OptorSim un CE può eseguire un solo job in un dato istante; la presenza di uno o più worker node all'interno del CE influisce sul tempo di esecuzione di un job, che diminuisce all'aumentare dei worker node.

Un punto importante da considerare riguarda il metodo di accesso ai file da parte dei job. I file che un job deve elaborare per portare a termine il proprio lavoro, vengono utilizzati

[1] Vedremo fra poco come il pericolo del deadlock non sussiste in questo semplice modello.

[2] Nel momento in cui è "down" alcune repliche dei propri file potrebbero subire delle modifiche, ed il sito in questione, essendo disconnesso dal sistema, non riceve tali aggiornamenti.

uno per volta, all'interno di un loop. Consideriamo un semplice meccanismo di locking, dove un job prima pone il lock su un file, poi esegue le sue elaborazioni ed infine rilascia il lock. Se il lock su quel file fallisce, in quanto un altro job aveva precedentemente ottenuto il lock sullo stesso file, si pone in attesa. In questo modo è impossibile generare situazioni di deadlock, in quanto non si possono generare dei cicli di job in attesa di lock.

Abbiamo scelto di implementare un meccanismo di locking basato sui logical file name. Quando un job deve accedere a un file per una operazione di scrittura deve prima ottenere il lock sul file logico, senza occuparsi delle varie repliche esistenti. Questo meccanismo è utile poiché in OptorSim un job elabora un file fisico (una replica) sempre attraverso il suo logical file names, e quindi è impossibile accedere a un file su cui un job detiene il lock.

L'uso del lock per operazioni di lettura è da valutare. Se ammettiamo la possibilità che un job possa leggere dei dati obsoleti (stale data), sebbene in un arco di tempo limitato, possiamo fare in modo che accessi in lettura non richiedano l'uso del lock sul file. Se un job esegue una operazione di scrittura su un'altra replica dello stesso file logico, le due operazioni vengono eseguite in parallelo, ed ovviamente il job leggerà dei dati che stanno per essere cambiati o lo sono già. Dovrebbe invece essere impossibile leggere e scrivere sulla stessa replica, ma questo fatto deve essere analizzato con maggior attenzione. La possibilità che un job legga dei dati obsoleti è una conseguenza della rilassatezza introdotta nel modello e le sue implicazioni dipendono dal contesto della simulazione, dal tipo di job che vengono eseguiti e dal contenuto dei file elaborati. Se questa rilassatezza non è accettabile possiamo fare in modo che anche le operazioni di lettura debbano richiedere il lock sul file, ma questo indurrebbe sicuramente un drastico calo delle prestazioni dato che nemmeno operazioni di lettura sullo stesso file logico potrebbero avvenire in parallelo. L'uso di due differenti tipi di lock, read-lock e write-lock, deve anch'esso essere valutato, specialmente riguardo al pericolo della starvation, che compare quando operazioni di lettura possono lavorare in parallelo; cosi‘ facendo infatti una operazione di lock potrebbe rimanere in attesa per un tempo indefinito.

La versione 0.5 di OptorSim non prevede una distinzione tra accessi ai file in lettura o scrittura. Chiaramente, per la valutazione del modello che stiamo definendo, questa distinzione è importante. La maniera più logica per effettuare questa distinzione sarebbe quella di modificare il file di configurazione dei job, al fine di contenere l'indicazione su quali file possono essere acceduti in lettura, scrittura o in entrambi i modi.

B.1 Implementazione

Vediamo adesso come è stato realizzato questo modello, esaminando prima alcuni scenari tipici che possono far comprendere le funzionalità messe a disposizione e dopo alcuni dettagli implementativi.

B.1.1 Scenari

Come abbiamo detto in OptorSim un solo job può essere in esecuzione in una dato CE in un dato istante; da questo punto in poi quindi i termini job e CE indicheranno la stessa cosa e saranno usati alternativamente. Inoltre useremo anche il termine file per indicare un file logico, riferito tramite il suo lfn. Con il termine replica invece intendiamo un file fisico riferito dal suo pfn.

Nel primo scenario mostrato in figura B.1 un job deve accedere e modificare un file, di nome lfn. Prima di tutto il job deve richiedere al Replica Consistency Service (RCS) il lock su quel file. In questo scenario il file non è "lockato" ed il RCS, dopo aver controllato il suo catalogo interno, concede il lock al job richiedente. A questo punto il job richiede al Replica Optimization Service (Optor) la miglior replica di quel file sulla quale eseguire le operazioni. Reperita la best replica, il job deve comunicarla al RCS. Adesso il job è libero di svolgere il proprio lavoro su questa replica. Quando ha finito il job deve dire al RCS di propagare gli aggiornamenti appena fatti alle altre repliche, dopo di ché rilascia il lock sul file e termina la propria esecuzione[3].

Il secondo scenario (figura B.2) è leggermente più complicato. Quando il Job1 richiede il lock sul lfn, il RCS rileva che un lock è già stato assegnato sul medesimo file ad un altro job, il Job2, e così non può concedere il lock al Job1. La risposta alla richiesta di lock è quindi negativa, e a questo punto il Job1 si pone in attesa e il RCS lo assegna alla coda relativa al lfn (vedremo fra poco come è organizzato il catalogo e come sono gestite le code dei job). Quando il job che detiene il lock sul lfn ha finito di fare il suo lavoro, rilascia il lock sul lfn e il RCS può notificare il Job1, che si sveglia ed effettua di nuovo la richiesta di lock sul lfn. Adesso il lock viene concesso e lo scenario si riconduce a quello precedente.

Per quanto riguarda operazioni di lettura senza richiesta di lock, i passi che un job deve eseguire sono i soliti che faceva in assenza del RCS. Con un lock semplice invece gli scenari sono uguali a quelli appena visti.

B.1.2 RCSim e il Replica Consistency Catalog

Questo modello di Replica Consistency Service è sviluppato come un componente indipendente e centralizzato, RCSim, che offre dei servizi e ne richiede altri. Nelle nostre simulazioni esso viene contattato direttamente dai job che richiedono e rilasciano i lock, chiedono di essere messi in coda per aspettare dei file e richiedono di propagare gli aggiornamenti alle altre repliche. Alcune di queste richieste il RCS le soddisfa direttamente,

[3] In realtà non termina, ma passa ad elaborare, se esiste, il file seguente.

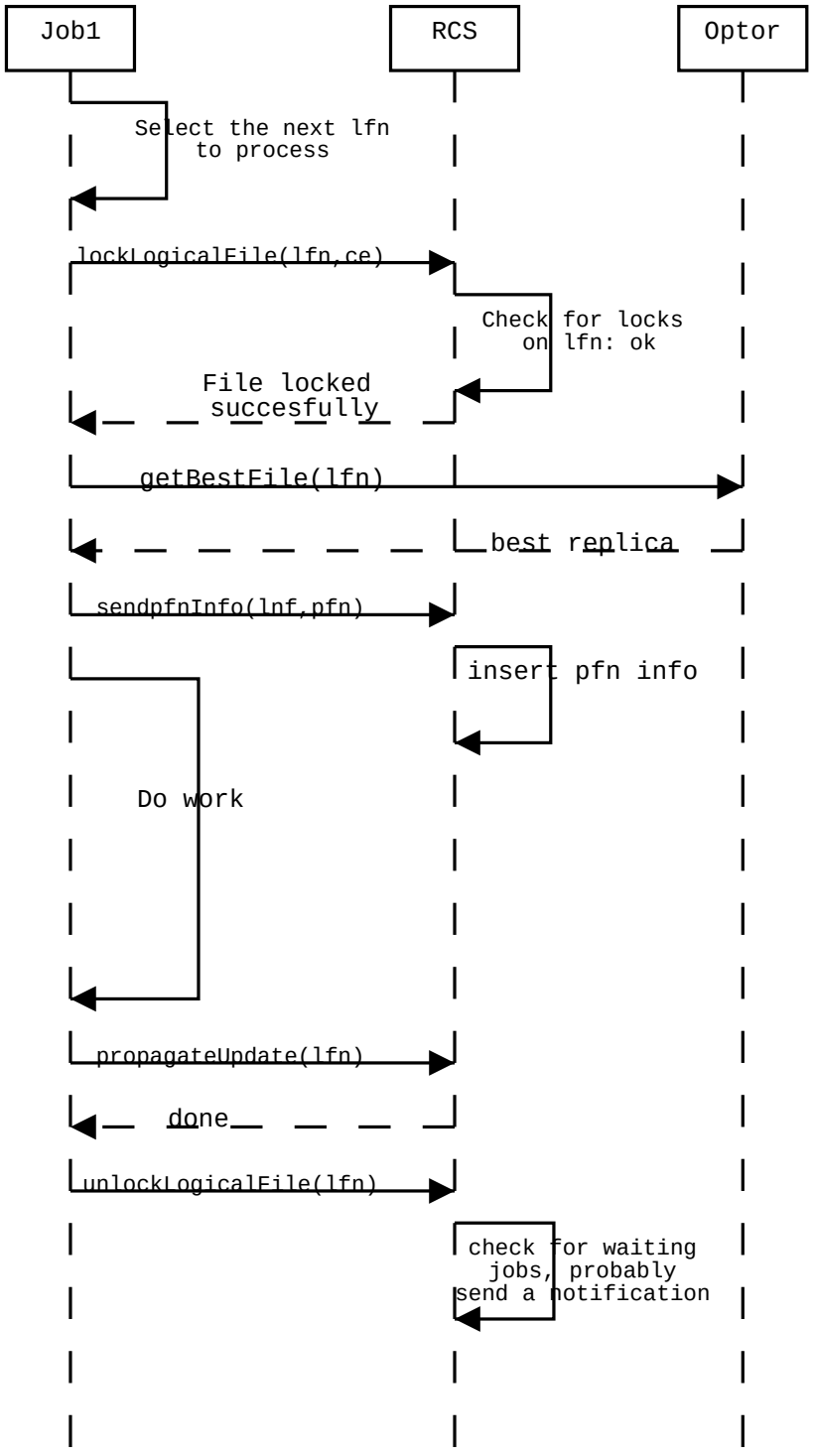


Figura B.1: Scenario 1

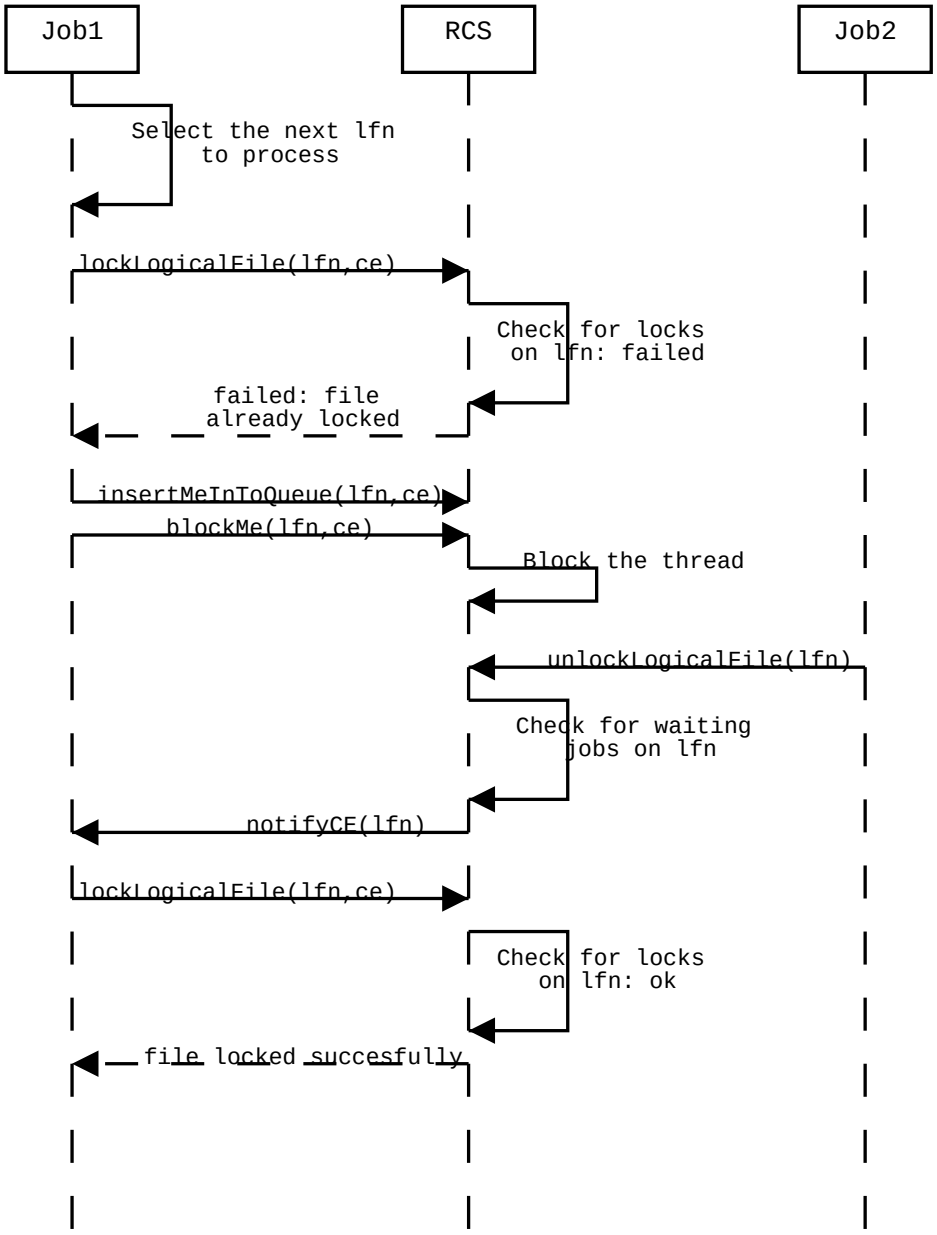


Figura B.2: Scenario 2

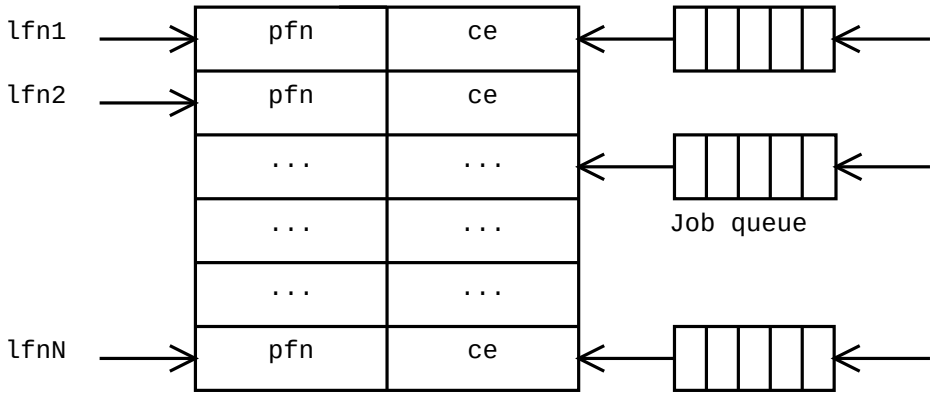


Figura B.3: Replica Consistency Catalog

effettuando delle elaborazioni sul proprio catalogo (Replica Consistency Catalog), per altre invece deve sfruttare i servizi offerti dal Replica Manager, Reptor, che in OptorSim viene simulato dal componente ReptorSim.

Il catalogo usato da RCSim contiene una lista di lfn, che sono quelli che in quel momento sono “lockati”. Per ogni lfn viene memorizzato anche il CE che detiene il lock, e la replica utilizzata. Quest’ultima informazione è fondamentale in quanto gli aggiornamenti da propagare sono proprio contenuti in questa replica. Ad ogni lfn è inoltre associata una coda, gestita con logica FIFO, in cui vanno a finire i job che sono in attesa di ricevere il lock sul medesimo file.

B.1.3 Diagramma delle classi

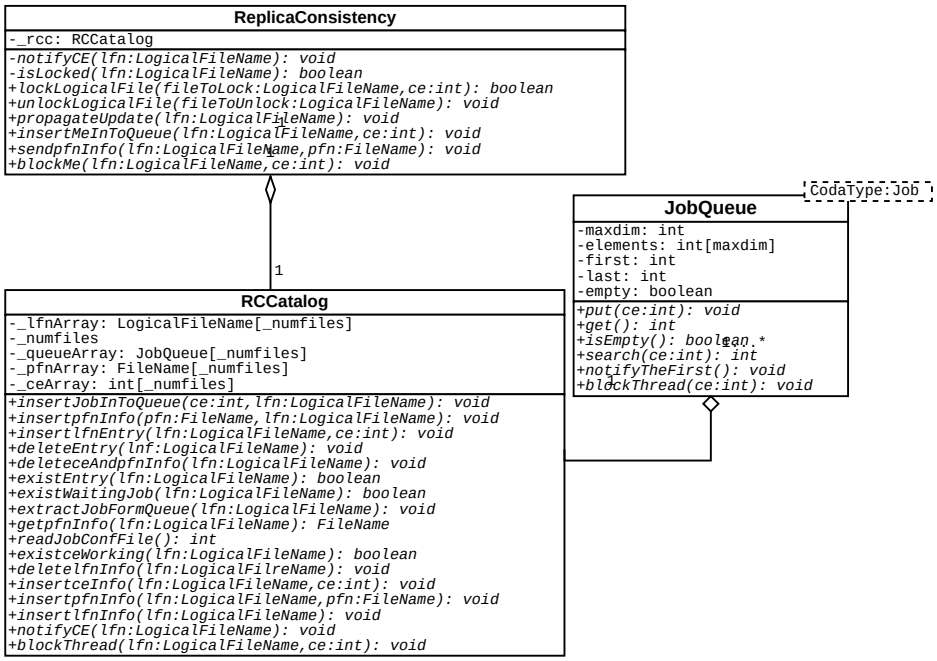


Figura B.4: Diagramma delle classi di RCSim

Bibliografia

[1] Open Grid Service Architecture. http://www.globus.org/ogsa/

[2] The Earth System Grid Site. http://www.earthsystemgrid.org

[3] The FusioGrid Site. http://www.fusiongrid.org

[4] The Global Grid Forum Site. http://www.ggf.org

[5] The Globus Replica Catalog. http://www.globus.org/datagrid/ replica-catalog.html

[6] The Globus Site. http://www.globus.org

[7] The Gome Site. http://auc.dfd.dlr.de/GOME/

[8] The Grid Physics Network Site. http://www.griphyn.org

[9] The GridFTP Protocol and Software. http://www.globus.org/datagrid/gridftp.html

[10] The GridPP Project Site. http://www.gridpp.ac.uk/

[11] The NASA's Information Power Grid Site. http://www.ipg.nasa.gov

[12] The NEES Consortium, Inc. Site. http://www.nees.org

[13] The NPACI Site. http://www.npaci.edu

[14] The Particle Physics Data Grid Site. http://www.ppdg.net

[15] Data Management (WP2) Architecture Report – Design, requirements and evaluation criteria. Technical Report DataGrid-02-D2.2-0103-1_2, DataGrid WP2, 2001. Draft.

[16] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronica Nefedova, Darcy Quesnel, and Steven Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing, 2001.

[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] William H. Bell, David G. Cameron, Ruben Carvajal-Schiaffino, A. Paul Millar, Kurt Stockinger, and Floriano Zini. OptorSim v0.5 installation and user guide. Technical report, 2003.

[19] P.A. Bernstain, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[20] A. Domenici, F. Donno, K. Pashen, G. Pucciani, H. Stockinger, and K. Stockinger. Replica Consistency in a Data Grid. In *9th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT03), Tsukuba, Japan*, 2003.

[21] A. Domenici, F.Donno, G. Pucciani, H. Stockinger, and K. Stockinger. Replica Consistency Service - Design Principles and Basic Architecture v0.3. Technical report, 2003. Draft.

[22] EDG WP6. *EDG User's Guide*, 2003.
`marianne.in2p3.fr/datagrid/documentation/EDG-Users-Guide.html`

[23] A. Chervenak et al. Giggle: A framework for constructing scalable replica location services. In *Proceedings of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002), Baltimore, USA*, 2002.

[24] L. Guy et al. Replica management in data grids. Working draft, Global Grid Forum (GGF4), Toronto, Canada, 2002.

[25] I. Foster and C. Kesselman, editors. *The Grid – Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[26] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91, 1998.

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

[28] David K. Gifford. Weighted voting for replicated data. In *ACM SIGOPS Symp. on Operating System Principles, Pacific Grove, USA*, pages 150–162, 1979.

[29] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Sasha. The dangers of replication and a solution. In *ACM SIGMOD Conference, Montreal, Canada*, pages 173–182, 1996.

[30] IBM. Ibm db2 data propagator site. `http://www-3.ibm.com/software/data/dpropr/index.html`

[31] J.Gray and A.Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1st edition edition, 1993.

[32] The MONARC Project. The monarc project. `http://monarc.web.cern.ch/MONARC/` (Project homepage).

[33] Yasushi Saito. Optimistic replication algorithms. 2000. unpublished.

[34] H. Stockinger, G. Pucciani, and F. Donno. Replica Consistency Service (RCS) - Interface v0.3. Technical report, 2003. Draft.

[35] Sybase. Sybase replication server site. `http://eshop.sybase.com/products/middleware/replicationserver`

[36] EDG WP2. Job description language how to. Technical Report DataGrid-01-TEN-0102-0$_2, DataGridWP1, 2001. Draft.$

[37] EDG WP2. Design of a replica optimization framework. Technical Report DataGrid-02-TED-021215, DataGrid WP2, 2002.

[38] EDG WP2. (WP2) Replica Manager – Design and API Specification. Technical Report DataGrid-02-ReplicaManager, DataGrid WP2, 2002. Draft.