Università degli studi del Piemonte Orientale "Amedeo Avogadro"

Facoltà di Scienze M.F.N.

Tesi di Laurea in

Informatica dei Sistemi Avanzati e dei Servizi di Rete

Studio ed implementazione di metodi di previsione dei guasti per politiche di scheduling in ambito Desktop Grid

Relatore
PhD. Massimo Canonico

Candidato
Guido Vicino

Anno Accademico 2006/2007

Ringraziamenti

Si ringrazia il PhD. Massimo Canonico
ed il Gruppo di Sistemi Distribuiti del Dipartimento di Informatica
per l'aiuto e l'interesse mostrato durante il lavoro di tesi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

In questo capitolo si descrive il modello computazionale nel quale si inserisce il lavoro di questa trattazione, ossia il *Grid Computing*. Dopo averne discusso brevemente la storia e gli obiettivi, si descrivono l'anatomia e gli aspetti tecnologici che lo caratterizzano ed una serie di applicazioni in campo economico e scientifico. Discussi questi aspetti più generali vengono esposti i problemi principali che questi sistemi devono affrontare e le soluzioni proposte dagli esempi più interessanti di queste infrastrutture hardware e software. Nel capitolo successivo vengono invece trattati direttamente il lavori fatti nella ricerca relativi al miglioramento dell'affidabilità e della velocità di risposta di questi sistemi, sfruttando diversi metodi di previsione statistica.

1.1 Grid Computing

Il *Grid Computing*, secondo la definizione data da Ian Foster e Carl Kesselman [24], è l'infrastruttura hardware e software che fornisce un accesso affidabile, consistente e pervasivo a grandi capacità computazionali. Questa definizione viene arrichita durante gli anni fino a specificare che il Grid Computing concerne la condivisione coordinata di risorse e la risoluzione di problemi in organizzazioni virtuali dinamiche e multi-istituzionali [23].

Il Grid Computing mira ad estendere i limiti del singolo calcolatore mettendo in collaborazione la grande varietà di risorse di memorizzazione e computazione disponibili all'interno delle sempre più veloci reti di calcolatori sparse per il mondo. Il Grid Computing prende in prestito il nome dal sistema di fornitura elettrica americano, nel quale ogni singolo utente abbonato può ricevere energia collegandosi alla rete di distribuzione, chiamata in gergo "grid". Nell'ambito informatico sia ha invece una rete di elaboratori o strumenti scientifici, ciascuno di essi dotato di diverse caratteristiche e da

una diversa tipologia di utenti. L'insieme dei calcolatori, delle risorse, della rete e del software che serve a gestirne la coordinazione e la distribuzione dei servizi all'utente viene chiamato analogamente *Grid*.

La Grid, come da definizione, si configura come un'infrastruttura hardware e software che permette la condivisione dinamica di risorse eterogenee appartenenti ad entità amministrative differenti tra loro. Le risorse possono essere gruppi dispersi di macchine, server, sistemi di memorizzazione, dati e reti che vengono integrate, gestite e condivise in un unico sistema sinergico. L'utente deve accedere in modo trasparente alle potenzialità del sistema disinteressandosi di come questo è configurato. È quindi presente un opportuno strato software che si preoccupa di nascondere e gestire autonomamente i dettagli relativi alla molteplicità delle risorse, all'autenticazione, all'accesso ai calcolatori o relativi all'allocazione dei lavori, delle applicazioni ed al trasferimento dei dati. Questo strato viene inserito tra l'utente e la rete di calcolatori prendendo il nome di *middleware*. L'illusione che si viene a creare è quella di un grande calcolatore distribuito in cui la potenza di memorizzazione e di calcolo è maggiore rispetto a quella della singola macchina. La Grid rientra quindi nella categoria dei sistemi distribuiti, in quanto insieme di componenti eterogenei e distribuiti in locazioni differenti,

Un concetto importante nel Grid Computing è quello di *Virtual Organization* (VO), sotto questo termine si identificano quegli individui e/o quelle istituzioni che sono definite da regole comuni di condivisione delle risorse. Queste regole sono dinamiche nel tempo e stabilite all'interno delle singole entità che formano una VO.

É necessario al fine di comprendere l'anatomia del Grid Computing odierno parlare di come esso si sia evoluto durante gli anni a partire dai primi modelli e dalle prime sperimentazioni. Nella prossima sezione si illustrerà quindi qual'è la storia del Grid Computing e quali sono stati i passi più rilevanti ed interessanti della sua evoluzione fino ai sistemi attuali.

1.2 Storia del Grid Computing

La storia del Grid Computing inizia nei primi anni '80 e '90 dove l'enfasi e l'attenzione erano sull'elaborazione parallela, vista sotto l'aspetto degli algoritmi e delle architetture. L'idea era quella di estendere il *parallel computing* tradizionale tramite programmi che utilizzavano "calcolatori paralleli virtuali", composti da sistemi di calcolo autonomi ed eterogenei al posto che architetture multiprocessore dedicate ed utilizzate all'interno della singola organizzazione.

Questo approccio inizialmente mirava a risolvere i problemi pratici che

insorgevano nell'accesso a risorse (non solo di calcolo) eterogenee e distribuite su scala geografica. Iniziarono così negli Stati Uniti i primi "progetti pilota" concepiti per studiare le possibili architetture e per scoprire l'effettiva utilità di tali sistemi (allora utilizzati esclusivamente per la ricerca scientifica) [28].

I primi antenati della Grid furono chiamati con il nome di *metacomputer* e comparirono verso il 1990 nell'intento di realizzare un'interconnessione dei centri di supercalcolo americani potenziandone le possibilità computazionali. I due progetti più importanti di quella che viene definita la prima generazione di sistemi Grid sono entrambi d'origine statunitense e nacquero nel 1995 con i nomi di *Factoring via Network-Enabled Recursion (FAFNER)* [41] e di *Information Wide Area Year* (I-WAY) [16].

Il *FAFNER* nacque come progetto il cui obiettivo era la fattorizzazione di numeri molto grossi, problema di rilevante importanza nel campo della sicurezza e della crittografia digitale. L'innovazione consisteva nel fattorizzare questi numeri tramite calcolatori distribuiti sulla rete. Molte delle tecniche, qui utilizzate per la suddivisione di grossi problemi computazionali in parti, furono poi sfruttate da middleware successivi come per il famoso *SETI@home* [4].

L'*I-WAY* era un progetto che mirava a collegare supercalcolatori utilizzando però le reti esistenti senza costruirne di nuove o dedicate. Vennero aggregate risorse appartenenti a ben 17 diversi enti americani, furono sviluppate oltre 60 applicazioni e fu creato un primo esempio di middleware Grid che forniva servizi per l'accesso alle risorse, il controllo, l'autorizzazione e il coordinamento tra di esse. Lo sviluppo di questi progetti ebbe un enorme importanza per la comprensione delle problematiche da affrontare al fine di offrire una piattaforma di calcolo effettivamente utilizzabile.

La seconda generazione di sistemi Grid fu orientata alla soluzione di quei problemi che vennero lasciati volutamente irrisolti dai progetti della prima generazione. Tali problemi erano ad esempio relativi alla gestione della eterogeneità delle risorse e all'incremento della scalabilità. Altro punto d'interesse era il miglioramento delle capacità di adattamento alla dinamicità delle risorse e delle applicazioni. Nell'autunno del 1997 ad un workshop tenuto all'*Argonne National Laboratory* nacque il progetto *The Grid*, gestito dai due ricercatori Ian Foster e Carl Kesselman. A questa coppia e al loro articolo "Globus: a Metacomputing Infrastructure Toolkit" [21] si deve la creazione di uno dei più importanti e famosi middleware della storia di questa tipologia di sistemi distribuiti. *Globus* infatti non fu solo il middleware di The Grid ma divenne in pratica lo standard "de facto" per la costruzione di soluzioni Grid Computing. Altro progetto rilevante della seconda generazione fu *Legion* [29], iniziato nel 1994 presso l'Università della Virginia. Legion si presentava come un metasistema operativo *object oriented*, dove ogni entità

era rappresentata da un oggetto (i file, i calcolatori sulla rete, etc.) dotato di specifiche procedure d'accesso e costituente una unica macchina virtuale estesa. La creazione di tali sistemi portò alla nascita di nuovo campo di ricerca nell'ambito dell'allocazione dei lavori computazionali e furono creati sistemi appositi per applicazioni Grid quali *Condor* [43], *PBS* [30], *SGE* [38] e *LSF* [48].

La ricerca sui sistemi Grid, sebbene nata negli Stati Uniti, crebbe anche in ambito Europeo dove si assistette ad esempio alla nascita di sistemi quali *GridWay* [31], *GridLite* [42] ed *Unicore* [19]. Si sentì quindi l'esigenza di tenere traccia e di coordinare i numerosi progetti di ricerca che a volte nascevano e crescevano parallelamente in vari posti ed organizzazioni del mondo. Tale sforzo si concretizzò nella formazione del *Global Grid Forum* (GGF), che è l'organismo di riferimento per lo sviluppo di omogeneità e standard dei protocolli all'interno dell'ambito Grid. Il risultato principale è stata la definizione degli standard *Open Grid Services Architecture* (OGSA), che mirano ad un'architettura che integri il paradigma Grid con i servizi web mediante i cosiddetti "Grid Service". Nel 2004 è stato inoltre definito il *Web Services Resource Framework* (WSRF), che consiste in una serie di specifiche per aiutare gli sviluppatori a scrivere applicazioni che utilizzino le risorse all'interno delle Grid.

La terza generazione di sistemi di Grid Computing, cioè quella attuale, mira a creare sistemi autonomi che siano in grado di configurarsi e riconfigurarsi in maniera indipendente per reagire ai cambiamenti dell'ambiente in cui operano. Sistemi capaci di ottimizzare il proprio comportamento in modo tale da raggiungere gli scopi prefissati, capaci di ripararsi da soli nel caso di malfunzionamenti e di proteggersi da attacchi esterni. Tali capacità vengono identificate come capacità di *self-autonomicity*. La ricerca odierna mira a realizzare tutto questo e a fornire una visione delle Grid come architetture orientate ai servizi *Service Oriented Architectures* (SOA) [28]. Questo termine identifica un'architettura software mirata al supporto dei requisiti dei processi aziendali e degli utenti tramite l'utilizzo di servizi indipendenti ai quali si può accedere in maniera trasparente.

Nella sezione 1.3 si descrivono quali sono state le architetture ideate durante questi anni, e quale modello i ricercatori hanno cercato di seguire nei loro progetti. Nella sezione 1.4 si illustrano invece quali applicazioni in campo scientifico ed economico si sono trovate al Grid Computing.

1.3 Architettura e middleware

L'architettura di una Grid è assai complessa e per essere descritta si fa spesso uso di una suddivisione a "livelli", ciascuno dotato di una funzione specifica. Analogamente ad altri paradigmi di questo tipo, nello strato più alto le funzioni saranno mirate all'utente, mentre nello strato più basso saranno localizzate l'hardware, i calcolatori e le reti (vedi Figura 1.1).

Figura 1.1: Anatomia di una Grid (Cern GridCafé [28])

In una Grid il livello più basso è rappresentato dalla *rete*, che assicura la connettività tra le risorse presenti. Sopra ad esso vi è il livello delle *risorse*, formato da tutte le risorse che attualmente compongono la Grid, quali computer, sistemi di memorizzazione, cataloghi di dati digitali ed anche sensori e strumentazioni scientifiche quali microscopi e telescopi connessi in rete. Il livello di *middleware* fornisce gli strumenti che integrano e gestiscono globalmente i vari elementi (server, memorizzazione e reti) all'interno di un unico ambiente Grid. Il middleware è la componente più importante da progettare in una Grid perché costituisce il suo cervello dove vengono implementate le politiche di accesso, autorizzazione, collaborazione e sicurezza di tali sistemi distribuiti. Al livello più alto si trova il livello delle *applicazioni*. Gli utenti qui interagiscono con le varie applicazioni scientifiche, ingegneristiche, economiche finanziare, mentre amministratori e sviluppatori si occupano di gestire i portali e gli strumenti di sviluppo. È possibile inoltre individuare un altro livello conosciuto come *serviceware*, che in alcune architetture e sistemi si occupa di fornire tutta una serie di funzioni amministrative per misurare

l'utilizzo della Grid da parte degli utenti, al fine di stabilire una "economia" interna delle risorse presenti in essa.

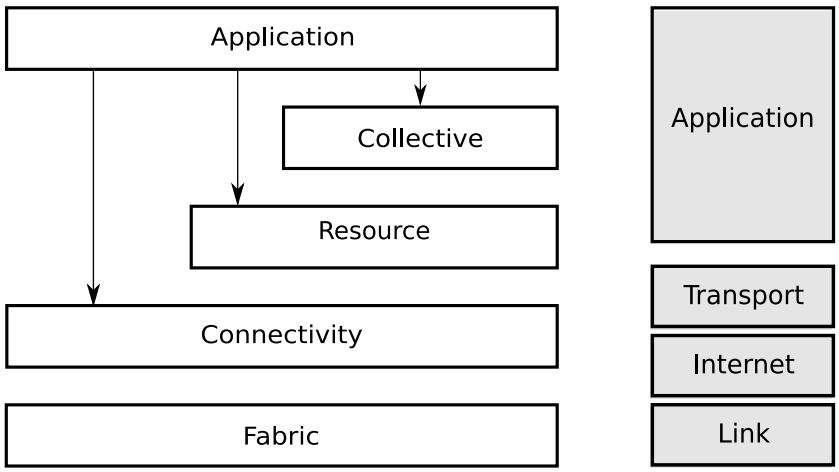

Figura 1.2: Anatomia di una Grid (I. Foster, C. Kesselman [23])

Questa appena descritta è una delle possibile visioni a livelli che sono state individuate dalla ricerca. Una seconda visione proposta sempre da I. Foster e C. Kesselman (vedi Figura 1.2) [23] identifica l'infrastruttura fisica della Grid con il termine inglese *Fabric*. Mentre il livello di middleware viene suddiviso in tre parti relative alla connettività (*Connectivity Layer*), alle risorse (*Resource Layer*) e alla coordinazione di quest'ultime (*Collective Layer*). Secondo questa visione il collo di bottiglia dell'architettura è rappresentato da un piccolo insieme di astrazioni e protocolli di trasmissione (qui identificati con i livelli Resource e Connectivity), sopra i quali i differenti livelli di strato più vengono poi strutturati.

Il livello *Fabric* fornisce dei percorsi predefiniti per l'interfacciamento alle risorse locali ossia implementa le operazioni di accesso alla Grid d'alto livello per mezzo di operazioni locali specifiche alle risorse. Fornisce quindi un insieme minimo di operazioni d'accesso per la richiesta, prenotazione e gestione delle risorse. Ad esempio riguardo le risorse computazionali questo livello deve fornire meccanismi per l'avvio, il monitoraggio ed il controllo dell'esecuzione dei programmi, meccanismi di gestione per il controllo dell'allocazione delle risorse, la loro prenotazione, funzioni d'interrogazione per determinare le caratteristiche hardware e software ed il loro stato corrente, e così via.

Il livello *Connectivity* definisce i protocolli centrali di comunicazione ed autenticazione richiesti per le transazioni sulla rete della Grid che devono

essere condotte in maniera semplice e sicura. I protocolli di comunicazione permettono lo scambio dei dati tra il livello Fabric e quello Resource, mentre quelli di autenticazione permettono meccanismi crittograficamente sicuri per verificare l'identità degli utenti e delle risorse. Vengono inoltre imposti dei requisiti alle comunicazioni che sono relativi al trasporto, l'instradamento e alla gestione dei nomi di dominio che attualmente nella maggior parte dei sistemi vengono offerti dai protocolli TCP/IP.

I requisiti che l'autenticazione di un sistema Grid sicuro deve seguire sono: il *single sign-on* senza il bisogno di autenticarsi manualmente le successive volte alle risorse definite in Fabric; la *delegazione*, ossia come i programmi acquisiscono i permessi utente; l'*integrazione* con le varie soluzioni locali di sicurezza ed infine le *user-based trust relationship*. Queste soluzioni devono fornire un supporto flessibile per la protezione delle comunicazioni e permettere ai proprietari di avere il controllo sulle decisioni relative alle autorizzazioni.

Questi due livelli appena descritti corrispondono ai livelli *Link*, *Internet* e *Transport* del protocollo Internet, i successivi tre saranno invece corrispondenti al livello *Application*.

Il livello *Resource*, costruito sopra i protocolli di comunicazione ed autenticazione forniti da quello Connectivity, definisce le regole relative alla negoziazione, iniziazione, monitoraggio, controllo, contabilizzazione e pagamento delle risorse individuali. Tutti questi protocolli hanno uno stretto rapporto con il livello Fabric al fine di poter comunicare e controllare localmente la singola risorsa. Si può fare una suddivisione tra le due classi principali di protocolli, ossia d'"informazione" e di "gestione". I primi si occupano di ottenere le informazioni relative alla struttura e allo stato di una risorsa. I secondi sono utilizzati per negoziare l'accesso ad una risorsa condivisa, per specificare i requisiti a tale risorsa e per le operazioni che istanziano relazioni di condivisione.

A differenza del livello precedente che si concentra sulle singole risorse, il livello *Collective* fornisce i protocolli ed i servizi che non sono associati con nessuna specifica singola risorsa, ma bensì sono relativi alla coordinazione e l'utilizzo combinato tra di esse.

Tali protocolli si occupano di un gran numero di servizi quali quelli relativi all'indicizzazione delle risorse, all'allocazione dei lavori, alla diffusione delle informazioni, alla monitorazione e diagnosi del sistema, alla replicazione dei dati, alla programmazione e collaborazione, alla gestione del carico, all'accounting e all'acquisizione delle risorse e così via. É facilmente intuibile come il "collo di bottiglia" precedentemente citato è costituito da questo livello e dal precedente, dovendo fornire e gestire un gran numero di servizi interoperanti tra loro.

L'ultimo livello, chiamato *Application*, è quello relativo alle applicazioni che operano all'interno di una organizzazione virtuale. In questo livello si devono fornire tutte le librerie che permettono di fare chiamate verso i servizi forniti dai livelli sottostanti. Per ciascun livello del nostro modello vi saranno delle *Application Programming Interface* (API), ossia delle librerie relative all'interfacciamento e l'utilizzo con ciascun elemento della struttura Grid.

Tramite questi oggetti è possibile lo sviluppo delle applicazioni che girano sulla Grid. Queste inizialmente erano legate ad ambiti scientifici ed accademici ma ora l'interesse è presente anche nei campi finanziari e commerciali, come discusso nella successiva sezione.

1.4 Applicazioni del Grid Computing

Lo spettro di applicazioni che possono essere eseguite su una Grid è piuttosto vario. Si hanno molteplici utilizzi, ad esempio si vanno a sfruttare in maniera opportunistica le risorse inutilizzate, si incrementa la potenza di calcolo e si permette l'esecuzione di applicazioni che richiedono risorse particolari non disponibili presso la singola entità. Altre possibilità sono la collaborazione remota tra persone, la condivisione dinamica delle risorse, il bilanciamento del carico tra di esse, e l'aumento dell'affidabilità tramite la ridondanza sia dell'hardware che del software.

Al fine di distinguere tra questo grande numero di applicazioni esistono diverse classificazioni possibili e diversi metri di misura secondo i quali queste possono essere differenziate. Si mostrano due classificazioni principali. La prima distingue tra *applicazioni tradizionali* ed *applicazioni grid-enabled* [24]. La seconda classe definisce cinque classi, differenziando a seconda dell'ambito applicativo e mostrando alcuni esempi chiarificatori [23].

La prima classificazione delinea due classi. Nella prima classe si troveranno le applicazioni tradizionali, create ed implementate senza un'ottica mirata all'utilizzo nell'ambito Grid. I lavori o *job* che si sottomettono sono caratterizzati quindi dal poter essere eseguiti su una singola risorsa. Questi job a loro volta possono essere di tipo sequenziale e richiedere un solo calcolatore per volta rispettante determinate caratteristiche, oppure possono essere paralleli e richiedere cluster o supercalcolatori. All'interno del middleware Grid vi deve essere quindi un sistema che cerca di adattarsi a questo tipo di applicazioni al fine di gestirne il lavoro sfruttando le risorse a disposizione. Questo normalmente viene fatto da un sistema denominato di *job management* che serve a sottomettere e gestire i job sulla Grid, un esempio famoso di questa soluzione è dato dall'accoppiata Condor-G e Globus [25].

Nella seconda classe si hanno invece applicazioni grid-enabled, ossia stu-

diate appositamente per utilizzare servizi di tipo Grid ed applicazioni che hanno bisogno di tale potenza di calcolo e memorizzazione per poter essere eseguite. Queste applicazioni nascono con lo scopo di sfruttare a pieno la Grid, conoscendone a priori le funzionalità e possibilità.

Un'altra classificazione necessaria riguarda i vari ambiti applicativi delle Grid e presenta le seguenti cinque classi [24]:

- le applicazioni di *Supercalcolo Distribuito (Distributed Supercomputing)* utilizzano le grid sostanzialmente per aggregare le risorse computazionali al fine di risolvere problemi che non possono essere studiati su un singolo sistema. A seconda del campo in cui si sta lavorando, queste risorse aggregate possono includere insiemi di supercalcolatori dislocati in posti distanti geograficamente o semplicemente includere i calcolatori usati all'interno di una compagnia. Alcuni esempi pratici includono: le simulazioni interattive distribuite che vengono usate per allenare e pianificare manovre militari; le simulazioni di processi fisici complessi richiedenti alte risoluzioni spaziali e temporali; i calcoli complessi per la cosmologia, la chimica computazionale o la modellazione climatica;

- negli ambienti di *Calcolo ad Alto Rendimento (High-Throughput Computing)* le Grid vengono utilizzate per allocare un grande numero di job debolmente accoppiati o indipendenti, con l'obiettivo di sfruttare i cicli di processore inutilizzati dei calcolatori. Per esempio questi sistemi sono stati utilizzati dall'azienda costruttrice di microprocessori AMD durante la fase di progettazione dei loro processori K6 e K7. Altre applicazioni sono state gli studi di sistemi complessi modellati tramite vincoli parametrici come quelli risolti tramite l'ausilio del sistema *Condor* oppure la risoluzione di problemi crittografici e crittoanalitici come nel progetto *Distributed.net*;

- le applicazioni di *Calcolo On-Demand* sfruttano invece le caratteristiche della Grid per fornire risorse in quegli ambiti dove possederle o comprarle direttamente diverrebbe una spesa troppo ingente. Si offre così l'opportunità di pagare o accedere gratuitamente alle risorse messe a disposizione da altre aziende o istituzioni. Le risorse possono essere di calcolo, software, memorizzazione, strumentazione scientifica e così via. Si hanno molte applicazioni sia dal punto di vista scientifico sia in quello commerciale: ad esempio, in questi ultimi anni alcune aziende hanno infatti puntato il loro business sull'offrire, tramite tali tecnologie, la loro "potenza" di calcolo ad aziende di tipo finanziario o farmaceutico;

- nel calcolo *Data-Intensive* si focalizza l'attenzione sul ricavare nuove informazioni da dati che risiedono su macchine geograficamente remote, da librerie digitali e da basi di dati relazionali. Questo processo di sintesi è intensivo dal punto di vista del calcolo e del trasporto dei dati. L'applicazione di una Grid forse più conosciuta è quella per gli esperimenti sull'alta energia condotti al laboratorio di ricerca europeo del CERN, ma anche le aziende di previsioni meteorologiche e gli ospedali che conducono analisi oncologiche sfruttano molto questi sistemi;

- il *Collaborative computing* racchiude quelle applicazioni che sono concepite principalmente per permettere e migliorare le interazioni tra persone. Tali applicazioni sono spesso strutturate in termini di spazi virtuali condivisi, e molte di esse mirano ad un utilizzo condiviso delle stesse risorse computazionali quali archivi dati e simulazioni.

  Si descrive ora un'applicazione rientrante nella quarta categoria sopra descritta conosciuta sotto il nome di *SETI@home* [4]. L'approccio di SETI@home è quello del *Public Resource Computing* che si basa sull'idea di sfruttare la capacità di calcolo e memorizzazione dati distribuita di un gran numero di computer messi a disposizione volontariamente da utenti presenti in Internet. Un altro nome comune dato a questa tipologia di sistemi è quello di *Desktop Grid* in quanto vengono eseguite normalmente su macchine di utenti e non macchine dedicate. Questa idea è stata lanciata per la prima volta dal progetto *Great Internet Mersenne Prime Search* (GIMPS) da Scott Kurowski nel 1996 ed al momento ancora attivo. L'obiettivo del progetto era scoprire numeri primi dalle caratteristiche particolari. L'idea era quella di offrire un piccolo client a vari volontari di Internet che avrebbero ceduto parte della capacità di calcolo dei loro elaboratori, quando questi erano inattivi. La stessa idea fu adottata dal progetto Seti@home, creato dal gruppo di ricerca della NASA che si occupa della ricerca di vita intelligente extra-terrestre chiamato *Search for Extra Terrestrial Intelligence* (SETI). L'obiettivo qui era quello di elaborare i dati recuperati dai segnali radio provenienti dallo spazio e distinguere tra quelli che possono essere prodotti da eventi naturali e tra quelli che non sono attribuibili ad essi. Per distinguere tra la grande varietà di segnali presenti vi era un grande bisogno di capacità computazionale. Il progetto proponeva all'utente di scaricare uno salvaschermo che installato sulla macchina ospite sarebbe partito nei momenti di inattività del computer mostrando una grafica accattivante e che contemporaneamente avrebbe scaricato da Internet piccole porzioni di dati di segnale da elaborare. Attualmente non sono stati ancora rilevati segnali di entità extraterrestri ma il

risultato a livello computazionale è stato rilevante: nel settembre del 2001 il progetto Seti@home ha raggiunto il ZettaFlow, corrispondente a 10^{21} operazioni in virgola mobile, corrispondenti a loro volta a 71 TeraFlop. La macchina più potente dell'epoca conosciuta con il nome di IBM ASCI White era capace in confronto di solo 12.3 TeraFlop. Grazie al grande successo di GIMPS e Seti@home sono nati diversi progetti analoghi fino alla realizzazione del progetto *BOINC* [2] che fornisce un ambiente generico per lo sviluppo di sistemi distribuiti volontari. Contemporaneamente sono stati sviluppati diversi middleware commerciali o di dominio pubblico con scopi analoghi a quelli dei progetti sopra descritti.

In questa sezione si è dicussa la grande varietà di applicazioni del Grid Computing, tra queste la maggior parte si occupano dello svolgimento di grossi "lavori" computazionali. Al fine di poter svolgere questi lavori al meglio, sono necessarie politiche e meccanismi che associno a ciascun lavoro la risorsa ad esso più congeniale. Nella prossima sezione si discuterà di quali sono le problematiche relative alla selezione delle risorse per le varie applicazioni.

1.5 Scheduling

In questa sezione parleremo dello *Scheduling* in ambito Grid, mostrando inizialmente le tassonomie che differenziano le varie politiche esistenti e presentando brevemente nelle varie sezioni gli algoritmi più famosi. Lo scheduling mira a trovare una giusta allocazione dei lavori o *job* da eseguire dato un numero variabile di calcolatori/risorse disponibili all'interno di una Virtual Organization. Il metro di misura che ci permette di definire se un algoritmo di scheduling è migliore di un altro può essere definito in diverse maniere ma dipende principalmente da quali aspetti e prestazioni si vogliono migliorare. Spesso si tende a favorire lo studio di algoritmi di allocazione che riducano il tempo di completamento, ma questa non è una regola generale.

Quando si costruisce un middleware per Grid è necessario pensare a come fornire delle buone prestazioni per lo scheduling e la terminazione dei job che sono in esecuzione su di essa. Spesso si è cercato di prendere in prestito gli algoritmi conosciuti per lo scheduling in ambito Cluster o Parallel Computing per i sistemi Grid. Tale scelta però spesso ha fornito dei risultati non soddisfacenti in quanto vi sono molte differenze tra i due campi. Questo perché i calcolatori ed i mezzi trasmissivi presenti in una Grid sono eterogenei e caratterizzati da forti dinamicità dovute a guasti e rallentamenti tipici di queste architetture. La grande produzione e varietà di algoritmi di schedu-

ling proposti non permette facilmente di definirne le prestazioni, è possibile però definire una tassonomia [11], [13] a seconda delle loro caratteristiche peculiari (Figura 1.3) .

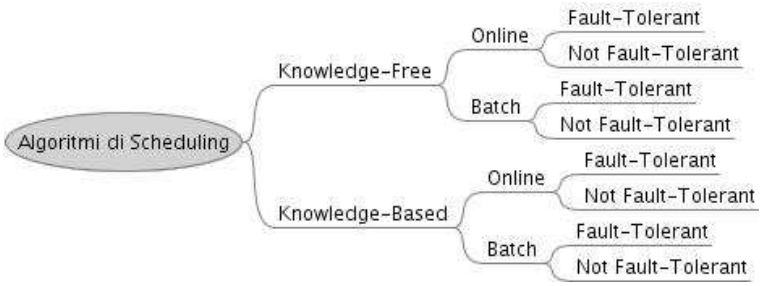

Figura 1.3: Algoritmi di Scheduling su Grid

Si distingue inizialmente tra algoritmi di tipo *Knowledge-based* o *Knowledge-free* che si differenziano a seconda del fatto che utilizzino o meno informazioni relative agli elaboratori presenti. L'aumento di informazione aiuta a migliorare le decisioni relative all'allocazione dei lavori, ma il recupero di queste informazioni può portare a rallentamenti e problemi nella gestione delle stesse. Quando si deve gestire l'allocazione di più lavori su diversi calcolatori si possono prendere due strade diverse. La prima è quella di selezionare un lavoro per volta e decidere, senza guardare gli altri lavori in coda, a quale macchina assegnarlo, le politiche che seguono questa strada vengono dette *Online*. L'altra strada è quella di selezionare un certo numero di job da assegnare, e poi creare associazioni tra questo campione di job ed i calcolatori attualmente disponibili, in questo caso le politiche sono dette *Batch*. L'ultima differenzazione riguarda invece la capacità o meno degli algoritmi di scheduling di tenere conto di eventuali guasti degli elaboratori o dei mezzi trasmissivi, differenziando così tra algoritmi *Fault Tolerant* o *Not Fault Tolerant*.

Nel descrivere vari esempi di questi algoritmi si è deciso di evidenziarli in quattro sezioni che riguardano le politiche online, batch, fault-tolerant ed una sezione conclusiva relativa ai modelli economici (rispettivamente le sezioni 1.5.1, 1.5.2, 1.5.3 e 1.5.4). All'interno di queste viene specificato quando le politiche descritte rientrano anche in altre tassonomie.

1.5.1 Politiche Online

La descrizione delle varie politiche online inizia da una politica molto semplice utilizzata in progetti Grid quali BOINC [2] ed XtremWeb [20] e conosciuta

come *First-Come-First-Serve* (FCFS) [35]. Questa semplicemente assegna il lavoro corrente alla prima macchina nella coda di quelle disponibili. Il risultato non è molto accurato dato che non si considera nessun tipo di informazioni riguardanti i lavori o i calcolatori ma si presta bene in ambienti estremamente eterogenei dove risulta estremamente difficile recuperare tali dati. Altri approcci sono stati ispirati dalle politiche di scheduling online esistenti in altri ambienti e si citano ora le più rilevanti.

Il *Minimum Completion Time* (MCT) [35] è un algoritmo che assegna ciascun job alla macchina che presenta per esso il tempo minimo di esecuzione. Questo implica che alcuni job verrano assegnati a calcolatori che non garantiranno il minimo tempo di esecuzione. Politica analoga è la *Minimum Execution Time* (MET) [35] che assegna ciascun job alla macchina che esegue quel job nel quantitativo minimo di tempo d'esecuzione. A differenza della politica precedente questa seconda non considera gli istanti in cui i calcolatori sono pronti ad iniziare la computazione. In questo caso vi può essere un uno squilibrio pesante nel carico lungo i calcolatori, però si ha il vantaggio di fornire a ciascun task la macchina che lo esegue nel minimo tempo d'esecuzione.

La politica *Switching Algorithm* (SA) [35] combina le due euristiche precedenti basandosi sulle seguenti osservazioni. La politica MET può potenzialmente creare disequilibrio nel carico lungo i calcolatori assegnando più job ad alcuni calcolatori rispetto ad altri, viceversa la MCT cerca di bilanciare il carico assegnando job con il minor tempo di completamento. Se i job vengono inviati in maniera casuale, è allora possibile usare la MET alle spese dell'equilibrio del carico fino ad una determinata soglia e poi utilizzare l'MCT per alleggerirlo. La SA utilizza quindi le politiche MCT e MET in una maniera ciclica a seconda della distribuzione di carico lungo i calcolatori.

La *K-Percent Best* (KPB) [35] considera soltanto un sottoinsieme di calcolatori m per la scelta di quella macchina che dev'essere assegnata al job. Il sottoinsieme è formato selezionando i primi $m \times (k/100)$ calcolatori, mentre il job è assegnato alla macchina che fornisce il minimo tempo di esecuzione in quel sottoinsieme. Se $k = 100$, allora si riconduce la KPB alla politica MCT. Se $k = 100/m$, allora si riconduce la KPB alla MET. Un "buon" valore di k associa un job ad una macchina entro il sottoinsieme formato dai calcolatori computazionalmente superiori.

L'ultima politica online alla quale si accenna brevemente è la *Opportunistic Load Balancing* (OLB) [35]. Lo scheduler assegna qui a ciascun job la macchina che diventa per prima disponibile, senza considerare il tempo di esecuzione del job su quella macchina. Se più calcolatori diventano disponibili contemporaneamente, ne viene selezionato uno in maniera arbitraria. Nonostante sembri un algoritmo estremamente semplice, una buona

implementazione risulta assai complessa in quanto lo scheduler deve esaminare tutti gli m calcolatori per trovare quello che diventa pronto per primo. Alcune implementazioni prevedono che siano gli stessi calcolatori inattivi a richiedere i job per se stessi accedendo ad una coda globale condivisa dei job.

1.5.2 Politiche Batch

Far corrispondere un numero di job indipendenti ad un insieme eterogeneo di computer è un problema che si conosce essere *NP Completo*. Altre politiche hanno invece come obiettivo primario la massimizzazione del throughput, in quanto questa è la misura di prestazione di un sistema predefinita negli ambienti orientati alla produzione. Le euristiche conosciute elencate offrono quindi solo soluzioni sotto ottimali. Spesso si tende a differenziare grossi gruppi di elaborazione da sottomettere chiamandoli job, e tra le varie unità indipendenti separatamente calcolabili detti *task*. L'insieme dei task che viene selezionato per lo scheduling viene chiamato *Bag of Task* (BoT) o *meta task* [6].

La selezione di questi BoT e lo scheduling di essi vengono effettuati dopo intervalli predefiniti, facendo uso principalmente di due strategie. La strategia *regular time interval* prevede di schedulare i meta task ad intervalli di tempo regolari, ad esempio ogni dieci secondi. L'unico caso in cui si ha una ripetizione avviene quando non arrivano più nuovi task dall'ultima associazione, oppure quando nessuno dei task precedentemente selezionati è completato. La strategia *fixed count* si basa invece su schedulare la BoT quando i task accodati superano un determinato numero, oppure quando tutti i task sono arrivati e quello attualmente in esecuzione conclude.

La prima politica batch citata si chiama *Min-min* [34]. Sia r_j il tempo atteso che la macchina m_j impiega per diventare disponibile ad eseguire il task dopo aver concluso l'esecuzione di tutti i task che gli sono stati assegnati fino a quel momento. Mentre e_{ij} rappresenta il tempo di esecuzione atteso del task t_i sulla macchina m_j nel caso questa sia libera quando t_i gli viene assegnato, mentre c_{ij} sono i relativi tempi di completamento. I primi c_{ij} sono calcolati utilizzando i valori di e_{ij} e r_j. Per ciascun task t_i, la macchina che fornisce il tempo di completamento minore è determinata dalla scansione della $i$esima riga della matrice d'utilità c (composta dai valori c_{ij}). Il task t_k che ha il minimo tempo di completamento atteso è determinato e poi assegnato alla macchina corrispondente. La matrice c e il vettore r (composto dai valori di r_j) sono aggiornati, ed il processo sopra descritto è ripetuto per tutti i task che non sono ancora stati assegnati ad una macchina.

La seconda politica discussa si chiama *Max-min* ed è simile alla precedente, differisce per il fatto che al ritrovamento per ogni task della macchina

che fornisce il tempo di completamento minimo, il task t_k che ha il *massimo* tempo minimo di completamento è determinato e poi assegnato alla macchina corrispondente. Questa seconda politica si comporta in maniera migliore in quei casi dove vi sono molti task corti rispetto a quelli lunghi.

La politica *Sufferage* [35], [11] si basa sull'idea che l'allocazione migliore può essere generata assegnando una macchina al task che dovrebbe "soffrire" di meno in termini di tempo di completamento atteso se quella particolare macchina non gli venisse assegnata. Il *valore di sofferenza* di un task t_i è dato dalla differenza tra il suo secondo minimo tempo di completamento (su una qualche macchina m_y) ed il suo primo tempo di completamento (su una qualche macchina m_x). Utilizzando m_x, risulta nel miglior tempo di completamento per t_i, ed utilizzando m_y nel secondo migliore. Da questa ultima politica si ottiene la politica di scheduling conosciuta con il nome di *XSufferage* che si prefigura come un estensione della precedente capace di sfruttare le località dei file utili ai task e già presenti sui calcolatori che li ospiteranno. In pratica se un file richiesto dalla computazione di un qualche task è già presente su un cluster remoto, quel task potrebbe "soffrire" se non assegnato ad un host in quel cluster. Il valore di sofferenza dovrebbe fornire una via semplice per catturare questa situazione ed assicurare il massimo riutilizzo dei file. Questo algoritmo presenta però il problema di aver bisogno di un certo numero di informazioni. Le misure richieste sono la velocità della macchina *HostSpeed*, il carico del processore medio dovuto ai processi locali *HostLoad*, ed infine il tempo di completamento di un task in un host con HostSpeed=1 e HostLoad=0. Questa politica è capace di fornire ottime prestazioni, ma necessita di un livello d'informazione relativo ai task e agli host troppo dettagliato che risulta irrealistico per molti ambienti di Grid Computing.

1.5.3 Politiche Fault-Tolerant

L'ultima classificazione possibile è relativa alla capacità o meno dello scheduler della grid di tenere in considerazione la possibilità di guasti ai calcolatori, in tal maniera si distingue tra scheduler che prevedono i guasti chiamati *fault tolerant* e tra quelli che non li prevedono chiamati *not fault tolerant*. I guasti ed i rallentamenti della rete e della computazione sono più gravi quando avvengono verso la fine dell'esecuzione di un applicazione sottomessa sulla Grid. Si utilizzano quindi come misure preventive principalmente le tecniche conosciute con il nome di *replicazione* e *checkpointing*.

Una delle soluzioni proposte si chiama *Workqueue with Replication* (WQR) [15] e viene usata nel middleware OurGrid [1]. Questa politica data una BoT seleziona un task in maniera arbitraria e lo assegna creando delle *repliche* su

diversi calcolatori. Quando una macchina finisce la computazione consegna il risultato all'utente e le altre repliche vengono cancellate. Tale politica ha il difetto di sacrificare cicli di CPU al fine di assicurarsi per il singolo task dei tempi di completamento adeguati nonostante i possibili guasti. Questa politica è stata poi estesa in maniera ulteriore tramite la *Workqueue with Replication Fault Tolerant* (WQR-FT) [6], in questo approccio si aggiunge alla replicazione anche dei meccanismi di *checkpointing* e di *automatic restart*. Il checkpointing è una tecnica che salva lo stato della computazione di un task ad intervalli predefiniti in maniera tale che nel caso avvenisse un guasto sull'host, il task potrebbe essere automaticamente inoltrato e riavviato automaticamente (automatic restart) su di una ulteriore macchina disponibile dall'ultimo punto di checkpoint salvato. Questo algoritmo ha fornito buone prestazioni in diverse sperimentazioni, l'autore ha dimostrato che WQR-FT non è solo capace di garantire il completamento di tutti i task in una Bag of Task, ma anche di presentarsi come un algoritmo knowledge-free capace di offrire prestazioni migliori delle alternative politiche di scheduling.

La politica *Distributed Fault-Tolerant Scheduling* (DFTS) [3] usa un certo quantitativo di informazioni per stimare il tempo di esecuzione dei task combinando questa informazione alla replicazione. In dettaglio, all'arrivo di un job, DFTS sceglie un insieme di n candidati e li prepara per l'esecuzione, ordinandoli secondo una stima dei loro tempi di completamento. Se la soglia desiderata di replicazione è maggiore che n, DFTS riserva un insieme di risorse uguale al numero delle repliche dei job non schedulati. Quando un job conclude con successo, DFTS invia messaggi a ciascuna locazione che ha precedentemente riservato al fine di poter riutilizzare tali locazioni per l'esecuzione dei job rimanenti. Questa politica knowledge-based ha come vantaggio di avere una decrescita lenta delle prestazioni nel caso di guasti ma ha il difetto di consumare anche cicli di computazione dovuti all'uso della replicazione.

La politica *Fault Tolerant Scheduling Algorithm* (FTSA) [44] è caratterizzata dall'uso di una *soglia minima di replicazione*. Vengono introdotti due valori, il numero n di replicazioni desiderato e la soglia di replicazione k, dove $k \leq n$. FTSA sceglie i migliori n calcolatori ordinati tramite una stima del tempo di completamento, ed invia a queste le repliche dei task. Il sistema deve assicurare che k repliche stanno girando. Cioè, il numero di repliche può scendere sotto n nel caso del fallimento delle stesse, ma non sotto k. Il problema principale di questa politica è definire dei valori di k ed n che forniscano risultati performanti.

1.5.4 Modelli Economici e Nimrod-G

L'idea intrinseca al Grid Computing è quella di condividere risorse tra più entità all'interno di un'unica organizzazione virtuale, gli algoritmi precedentemente presentati non si preoccupano di come gli utenti si rapporteranno ad essi, non ponendo volutamente regole o vincoli di alcun tipo che limitino l'utilizzo di determinate risorse, che possono essere sfruttate illimitatamente [27]. Il comportamento degli utenti relativamente all'ottenimento delle risorse può essere rappresentato tramite i classici modelli economici, questo con l'intento di limitare comportamenti negativi o antisociali all'interno di una Grid. Tipico è il fenomeno dei *free-ride*, ossia quando gli utenti consumano le risorse donate da altri utenti senza però a loro volta donarne loro. Molti sistemi come ad esempio OurGrid cercano di incentivare la riduzione di questo fenomeno e a collaborare con il sistema tramite diverse tecniche [5]. Un altro esempio potrebbe essere quello di un utente malevolo che decide di sottomettere e replicare grossi task al fine di rallentare i calcolatori presenti sulla Grid e danneggiare gli utenti che vi stanno lavorando. L'idea base è quella di associare ad ogni risorsa di calcolo, memorizzazione o altro un *costo d'utilizzo*, e per ogni utente o applicazione fissare un *budget* che gli permetta di "acquistare" servizi dai calcolatori presenti sulla Grid. Fissati questi dati si applicano modelli di mercato in maniera tale da avere dei vincoli e dei controlli sull'uso della Grid, queste regole saranno fissate da chi gestisce la Grid.

L'approccio forse più conosciuto a questa tipologia di sistemi è *Nimrod* [10]. Nimrod è stato sviluppato in Australia all'Università di Monash. Inizialmente nacque come un sistema per il calcolo parametrico distribuito di problemi lungo un insieme di risorse computazionali basandosi sull'approccio economico. Era possibile fissare e determinare per ciascun utente un budget disponibile e dare delle priorità ai job. I diversi studi mostrano che questo sistema funziona con successo per insieme statici di risorse computazionali ma che esibisce un certo numero di debolezze che lo rendono inutilizzabile quando adottato in un contesto dinamico di larga scala come quello Grid. Il problema è che in un ambito reale le Grid su larga scala esibiscono differenti proprietà:

1. i nodi sulla grid sono tipicamente sparsi lungo un certo numero di differenti domini amministrativi;
2. ciascun dominio ha la sua politica per l'allocazione delle risorse;
3. ciascun dominio ha un suo sistema per l'accodamento dei job, dei suoi personali costi d'accesso ed un quantitativo variabile di potere computazionale disponibile.

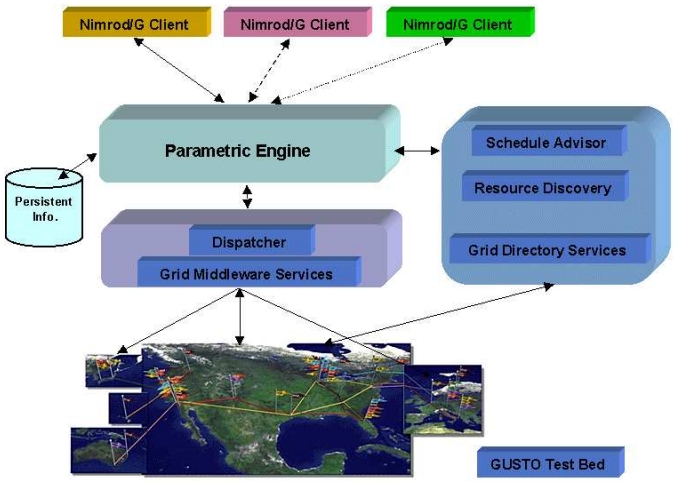

Figura 1.4: L'architettura di Nimrod-G (R. Buyya, D. Abramson, J. Giddy [10])

Per adattarsi a tali caratteristiche è stato sviluppato un nuovo sistema chiamato *Nimrod-G* [10] (Figura 1.4), il quale utilizza il middleware Globus per interfacciarsi con la Grid, e che risolve opportunatamente tali problemi.

Nimrod-G utilizza un *Job Wrapper* che è responsabile dello *staging* delle applicazioni e dei dati, cioè di preparare i calcolatori che ospiteranno i job, spostando i dati necessari e così via. Il job wrapper agisce anche come mediatore tra il motore parametrico e la macchina attuale su cui il task sta girando. La sua funzione primaria è di inviare i risultati dal client remoto indietro fino al motore centrale. Perciò è possibile monitorare le risorse utilizzate dal client remoto e riferire quei dati al sistema di contabilità delle risorse. Nimrod/G integra modelli economici nel suo sistema permettendo di schedulare job sulla base di scadenze e budget, potendo ad esempio formulare asserzioni quali:

- "per piacere completa questo task entro il giorno X e con un budget Y";
- "se puoi completare il task X entro il giorno Y io ti fornisco un compenso di Z".

Il sistema cerca di soddisfare i vari vincoli imposti minimizzando a seconda delle configurazioni i tempi o i costi. I prezzi di ciascuna risorsa vengono invece decisi dinamicamente utilizzando come strumento il *Grid Architecture for Computational Economy* (GRACE). Tale sistema è stato sviluppato per interagire e fornire un sistema all'economia computazionale con sistemi Grid

quali Globus o Legion fornendo librerie opportune per interfacciarsi con i vari sistemi.

L'unione tra economia computazionale e Grid Computing è stato spesso proposto ed implementato in diversi progetti, purtroppo sebbene i modelli utilizzati e studiati siano stati ben strutturati, si è pensato poco a come gli utenti reagiranno a tali sistemi e se sapranno sfruttarli a pieno. Tali esempi inoltre mirano a risolvere il problema dell'allocazione sotto un punto di vista "sociale" curando il comportamento degli utenti, ma disinteressandosi dei problemi relativi all'effettiva affidabilità delle risorse fornite.

1.6 Software per il Grid Computing

Esistono diversi software e middleware per realizzare sistemi di Grid Computing, alcuni di essi sono distribuiti sotto licenze gratuite, altri sotto licenze commerciali. Si mostrano tre pacchetti middleware interessanti che riassumono in parte le varie soluzioni e alternative presenti nel Grid Computing.

1.6.1 Globus Toolkit Version 4

Il *Globus Toolkit 4* (GT4) [22] è una collezione di librerie e programmi indirizzati alla risoluzione dei problemi comuni che occorrono quando si costruiscono servizi ed applicazioni per il Grid Computing. Il software fornisce una varietà di componenti e capacità che comprendono:

- un insieme di *implementazioni di servizi* che trattano la gestione delle risorse, i movimenti di dati, la scoperta dei servizi;
- strumenti per la costruzione di *servizi Web*, in Java, C e Python;
- una potente *infrastruttura per la sicurezza* basata su standard per l'autenticazione e l'autorizzazione;
- delle *librerie API per il client* (in differenti linguaggi) e programmi a riga di comando per l'accesso ai vari servizi e alle varie caratteristiche.

Inizialmente Globus è nato dalla necessità di creare un ambiente Grid per la gestione di organizzazioni virtuali in ambito scientifico. Ma recentemente le applicazioni commerciali hanno suscitato sempre più interesse fino a diventare molto importanti per questo progetto. Inoltre GT4 fa un uso estensivo di "Web Services" per definire le sue interfacce e strutturare i suoi componenti. Tali servizi forniscono flessibilità ed estensibilità ed usano i meccanismi basati su descrizioni in XML. Si è così facilitato lo sviluppo di

sistemi orientati ai servizi, fornendoli grazie ad un'architettura che permette applicazioni ed accessi sicuri tramite standard uniformi.

Le componenti dell'architettura di Globus possono essere riassunte in quattro categorie:

- *sicurezza*: si offrono i servizi di autenticazione, protezione e delegazione dei messaggi con la possibilità di single sign-on tramite certificazioni. Tali servizi sono offerti dalla *Grid Security Infrastructure* (GSI);

- *gestione dei dati*: si offrono i servizi per gestire e condividere la mole dei dati rispettando le politiche di gestione locali al fine di schedulare e catalogare in modo efficiente le risorse. Il sistema principale per il trasferimento dei dati messo a disposizione dal Globus Toolkit è il *GridFTP* che è una estensione del classico protocollo FTP. Inoltre vengono forniti sistemi per l'accesso e l'integrazione dei dati basati su modelli relazionali e descrizioni XML;

- *servizi d'informazione*: viene fornita un'infrastruttura di informazioni per l'indicizzazione delle risorse disponibili e lo stato della Grid. Quest'infrastruttura è fornita tramite un sistema chiamato *metacomputing Directory Service*, un servizio basato su LDAP, che può essere interrogato tramite vari strumenti come il recente *WebMDS* che può essere configurato tramite trasformazioni XSLT per creare viste specializzate dei dati memorizzati;

- *gestione dell'esecuzione*: implementata tramite il servizio *Grid Resource Allocation and Management* (GRAM) che offre un servizio Web per l'iniziazione, il monitoraggio e la gestione dell'esecuzione di computazioni arbitrarie su computer remoti. Si permette inoltre di specificare il tipo e la quantità di risorse desiderate, i dati che devono essere gestiti verso e dal sito di esecuzione, gli eseguibili ed i loro argomenti, le credenziali da usare.

Globus Toolkit, affiancato a sistemi quali Nimrod, è uno dei sistemi più usati per il Grid Computing, configurandosi come uno strumento molto potente, dov'è possibile integrare un grande numero eterogeneo di risorse. Questa sua potenza l'ha reso il sistema di riferimento nei gruppi di ricerca di tutto il mondo, rendendolo quindi uno standard di fatto di tali sistemi informatici. Lo svantaggio principale di Globus è l'essere costruito come un sistema piuttosto complesso che presenta una lenta curva d'apprendimento anche per l'amministratore più esperto che si trova a doverlo installare all'interno di un'organizzazione. L'amministrazione di Globus, la sua configurazione, gli

aspetti legati alla sicurezza e ai servizi non lo rendono un sistema di facile e veloce utilizzo. Negli anni si sono fatti passi avanti, ed alcune aziende e gruppi, hanno fornito meta-sistemi per la configurazione ed installazione veloce di questo prodotto.

Sebbene Globus Toolkit sia il punto di riferimento per la collaborazione tra grossi gruppi di ricerca, vi è l'esigenza di software più semplici e veloci all'interno delle piccole realtà. Alcune soluzioni sono state proposte dai sistemi di Public Resource Computing, citiamo ora due progetti di questo tipo.

1.6.2 Boinc

BOINC (Berkeley Open Infrastructure for Network Computing) [2] è un software che permette agli scienziati di creare ed operare molto semplicemente su progetti di Public Resource Computing [14], che come spiegato precedentemente sfruttano la potenza di calcolo degli elaboratori messi a disposizione volontariamente dagli utenti su Internet. Nato dalla spinta e dal successo del progetto SETI@home, questo strumento permette ai possessori di un computer connesso in rete di partecipare ai diversi progetti di ricerca che sfruttano BOINC e di specificare quanto delle loro risorse vengono allocate per questi progetti.

Il Public Resource Computing ed il Grid computing condividono l'obiettivo di fare un uso migliore delle risorse computazionali esistenti. Tra queste due categorie di sistemi vi sono però profonde differenze, ed i sistemi middleware per Grid Computing non risultano adatti ad un uso simile in quanto normalmente gestiscono risorse legate ad organizzazioni ed enti ben conosciuti. Tutte queste risorse sono gestite centralmente da professionisti dell'informatica che lavorano a tempo pieno per la loro gestione e per assicurarsi che siano connessi a tempo pieno. La differenza sostanziale nel Grid Computing è data quindi dalla relazione simmetrica che vi è tra le organizzazioni, infatti ciascun ente facente parte la VO condivide le risorse oltre che sfruttarle. Comportamenti maliziosi, quali la falsificazione intenzionale dei risultati, vengono puniti tramite meccanismi esterni al sistema. In contrasto, la condivisione pubblica di risorse volontaria coinvolge una relazione assimetrica tra progetto e partecipanti. BOINC si propone come un sistema creato per ridurre le barriere ed i costi di accesso alla partecipazione del Public Resource Computing o "volunteer computing". Questo significa che un ricercatore con una conoscenza media di informatica potrà creare ed operare su un grosso progetto di computazione con circa una settimana iniziale di lavoro. Al fine di incentivare ulteriormente i partecipanti è stato creato un sistema di ricompense tramite "crediti" che permette all'utente di visualiz-

zare numericamente qual'è stato il suo contributo e gareggiare con gli altri volontari tramite una "lista dei migliori volontari" pubblicata in rete.

Il sistema di server di BOINC è focalizzato attorno ad una base di dati relazionale che memorizza le descrizioni relative alle applicazioni, alle piattaforme, alle versioni, alle unità di lavoro, ai risultati, gli account e così via. Il server è strutturato come un insieme di servizi web e demoni: i *Server di scheduling* gestiscono le chiamate procedurali remote dai client, fornendo i risultati dei lavori completati. I *Data server* gestiscono l'upload dei file utilizzando un meccanismo basato su certificati in maniera tale da legittimare soltanto quei file che rispettino i limiti imposti relative alle dimensioni. Il download dei file viene invece gestito tramite semplice HTTP. I partecipanti entrano a far parte di un progetto riempendo un modulo di registrazione sul relativo sito web e scaricando il *Client BOINC*. Questo client può operare in maniere differenti, come uno screensaver mostrante i grafici relativi all'applicazione in esecuzione oppure come un servizio Windows, oppure come un applicazione, o un programma a riga di comando UNIX e così via. Una *Workunit* rappresenta l'input per una computazione ed è composta dall'applicazione, da un insieme di referenze legate ai file di input, da un insieme di istruzioni a riga di comando con i relativi parametri, ed infine dalle variabili d'ambiente. Ciascuna workunit ha dei parametri come i requisiti di computazione, memoria e storage ed una scadenza flessibile per il suo completamento. Un *Risultato* rappresenta il prodotto finale della computazione e consiste di una referenza alla workunit e di una lista di riferimenti ai file di output. Ogni file (associato ad una versione delle applicazioni, delle workunit o dei risultati) ha un un nome univoco ed immutabile lungo tutto il progetto. Ciascun file può essere replicato e porta con se tutti i riferimenti alle varie versioni da scaricare o sottomettere.

Quando il client comunica con il server di scheduling gli riporta il lavoro completato, e riceve un documento XML contenente una descrizione dell'entità sopra descritte. Il client subito dopo scarica e sottomette i file ed esegue la computazione; questo massimizza la concorrenza, utilizzando diverse CPU quando possibile e sovrapponendo comunicazione e computazione. Il sistema di calcolo di BOINC fornisce inoltre la possibilità di distribuire lo spazio di memorizzazione. BOINC è strutturato in maniera da effettuare computazione ridondante al fine di assicurarsi che nel caso di guasti od interruzioni arrivino sempre un certo numero N di risultati. Questo numero viene poi confrontato per identificare o rifiutare eventuali risultati erronei e tra tutti questi selezionare un possibile *risultato canonico*. Se questo non viene ottenuto, BOINC non crea risultati per quella determinata workunit e la risottomette fino ad un certo limite di tempo e di fallimenti.

Una limitazione di BOINC è quella di non poter integrare tra le sue risor-

se di calcolo strumentazioni e cluster già esistenti. BOINC risulta quindi la soluzione più utilizzata per il Public Resource Computing ed analogamente ad altri progetti gestisce l'affidabilità in modo knowledge-free sfruttando l'idea che la replicazione e la ridondanza della computazione offrano una buona sicurezza ed efficienza per un sistema di tale tipo.

1.6.3 OurGrid

OurGrid [1] è un progetto nato nel 2004 per la creazione di una grid computazionale aperta e di libero accesso di tipo Desktop Grid. L'idea è quella di fornire un sistema semplice e disponibile a tutti tramite il quale accedere ad una grossa quantità di risorse di calcolo. Questa potenza di calcolo viene ottenuta dalle risorse inattive di tutti i partecipanti e viene condivisa su Internet. La tecnologia usata è di tipo peer-to-peer in maniera tale che ogni laboratorio abbia interesse a condividere parte delle sue risorse per riceverne in cambio altre.

La piattaforma attualmente esistente permette di eseguire applicazioni i cui task non comunichino tra di loro durante l'esecuzione. Questa tipologia di applicazioni si chiama *Bag of Task* e permette quindi l'uso della Grid per sistemi di simulazione, data-mining e ricerca. Uno dei fattori più interessanti di OurGrid è la semplicità d'uso, chiunque possegga una macchina Linux, può installare e configurare tutte le sue componenti in pochissimo tempo.

L'architettura di OurGrid è composta da cinque componenti: *MyGrid*, *Peer*, *User Agent*, *GuM* e *Core Peer*. L'idea è quella di avere diversi domini amministrativi che mettono in condivisione dei calcolatori detti GuM. Su ciascuno di questi calcolatori gira una componente software detta User Agent che si occupa di tutte le funzionalità base relative alla strumentazione e alla gestione dei guasti. Per ogni dominio amministrativo vi è un coordinatore detto Peer, questo coordinatore ha due ruoli. Dal punto di vista di un utente il Peer si può vedere come un fornitore di calcolatori, dal punto di vista dell'amministratore esso invece determina come e quali calcolatori possano essere utilizzati e condivisi per l'esecuzione dei lavori sottomessi. Ogni persona o laboratorio che voglia accedere ai GuM presenti nel suo dominio amministrativo o in quello di altri deve utilizzare l'interfaccia utente chiamata MyGrid. Tramite questa interfaccia è possibile descrivere, eseguire e monitorare i propri job in esecuzione sui vari calcolatori. Questa componente non è solo un front-end ma svolge anzi un ruolo centrale effettuando infatti lo scheduling dei Job, recuperando la lista dei calcolatori disponibili dai vari Peer e sottomettendo i vari task ai GuM. L'ultima componente si chiama Core Peer. Il cui scopo è garantire che i nuovi Peer siano conosciuti dagli utenti della comunità di OurGrid. Se un Peer vuole condividere le proprie

risorse con la comunità e condividere le sue risorse con gli altri Peer, annuncia se stesso al Core Peer, e questo informa agli altri l'esistenza dell'ultimo arrivato.

La sicurezza su OurGrid è gestita tramite l'utilizzo della componente *Sandboxing Without A Name* (SWAN) che si occupa di proteggere gli host locali da codice malevolo. SWAN è una soluzione basata sulla creazione di una macchina virtuale tramite il software *Xen* [8] che isola il codice sottomesso dentro uno spazio protetto detto *sandbox*, dove all'interno di esso non è possibile accedere ai dati locali o alla rete.

Gli sviluppatori di OurGrid sono riusciti a creare un sistema facile e veloce, ma la natura stessa peer-to-peer di questo progetto porta ad alcuni problemi. Questo middleware non dispone di alcun sistema che miri a risolvere i comportamenti errati degli utenti quali quelli di free-riding; è possibile quindi per un utente malevolo creare rallentamenti ed occupare risorse senza alcun controllo.

Gli sviluppatori di OurGrid hanno deciso di utilizzare due politiche di scheduling, una è la Workqueue with Replication (QWR) spiegata precedentemente, e l'altra è una politica chiamata *Storage Affinity*. La prima politica come detto in precedenza è knowledge-free e quindi può portare ad una cattiva allocazione delle risorse. La seconda invece mira ad un riutilizzo dei dati per evitare i trasferimenti non necessari ed anche questa può portare ad allocazioni pessime delle risorse. Durante il lavoro di tesi, abbiamo provato ad estendere e creare altre politiche di scheduling estendendo il codice di OurGrid, purtroppo ci si è riusciti in parte e stravolgendo la natura del modello architetturale di alcune componenenti, gli sviluppatori infatti hanno creato un prodotto che nonostante sia modulare è mirato ad un utilizzo con politiche non knowledge-based. Un altra mancanza è quella di non poter definire vincoli parametrici complessi per la ricerca di calcolatori con caratteristiche specifiche, infatti ogni GuM è descritto all'interno di un descrittore conservato nello spazio di memorizzazione del Peer, ma la sintassi e le opzioni possibili sono molto limitate.

Nonostante questi "difetti" si presenta come un progetto utile dotato di una comunità molto partecipe che lo rende un progetto attivo ed in continuo sviluppo. Presenta inoltre il vantaggio della semplicità d'uso che lo rende facilmente fruibile a gruppi che non abbiano interesse ad usare progetti più complessi come Globus che richiedono buone conoscenze tecniche e lunghi tempi di amministrazione.

Capitolo 2

Predizione dell'affidabilità

Lo scopo di questa tesi è studiare i vari metodi per l'ottenimento di previsioni relative all'affidabilità delle macchine e determinarne l'accuratezza. Descriveremo in questo capitolo i cinque metodi di previsione che abbiamo ritenuto più interessanti ed importanti in questo ambito.

2.1 Predizione Lineare

La *predizione lineare* è una tecnica nata nell'ambiente dell'elettronica e serve a predirre il comportamento futuro dei segnali elettrici. Questa tecnica prevede un operazione matematica che, dato un segnale continuo, lo "discretizza" in una serie finita di valori temporali al fine di effettuare una stima dei valori futuri come funzione lineare dei campioni passati. Si sono trovate diverse applicazioni in campi differenti rispetto all'ambito in cui questa tecnica è nata, quali la neurofisica, la geofisica e l'analisi del parlato. Un riferimento importante ad un'applicazione nel campo del Grid Computing è dato da J. Mickens e B.D. Noble in [37] dove viene realizzato un predittore dell'affidabilità delle macchine tramite predizione lineare.

In questa sezione vengono illustrate le basi matematiche della predizione lineare al fine di chiarirne il funzionamento e l'applicazione nel campo del Grid Computing [36].

Nell'analisi dei segnali è possibile campionare ciascun segnale continuo $s(t)$ al fine di ottenere un segnale "discretizzato" $s(nT)$, che equivale ad una serie temporale dove n è una variabile intera e T è l'intervallo di campionamento. La frequenza di campionamento diviene quindi $f_s = 1/T$. D'ora in poi si userà per $s(nT)$ la dicitura s_n. É necessario per comprendere il comportamento futuro del segnale un modello parametrico che lo rappresenti. Questo modello è ottenuto considerando il segnale s_n come l'output di un

qualche sistema con un input sconosciuto u_n la cui relazione è data da

$$s_n = -\sum_{k=1}^{p} a_k s_{n-k} + G\sum_{l=0}^{q} b_l u_{n-l}, \qquad b_0 = 1 \tag{2.1}$$

dove $a_k, 1 \leq k \leq p, b_l, 1 \leq l \leq q$, ed il guadagno G sono parametri relativi al sistema ipotizzato. Quando l'input del sistema è sconosciuto allora si può approssimare la precedente equazione e predirre il segnale s_n come soltanto la somma pesata dei campioni passati

$$\tilde{s}_n = -\sum_{k=1}^{p} a_k s_{n-k} \tag{2.2}$$

Allora l'errore tra il valore vero s_n e quello predetto $\tilde{s}_n$ è dato da

$$e_n = s_n - \tilde{s}_n = s_n + \sum_{k=1}^{p} a_k s_{n-k} \tag{2.3}$$

dove e_n è anche conosciuto come il *residuo*.

L'intento è quello di stimare quindi i parametri a_k in modo tale da poter poi predirre i valori futuri. Per fare questo viene utilizzato il *metodo dei minimi quadrati*. Tale metodo suppone che fornito un insieme di campioni (x_i, y_i) dove $i = 1, 2, ..., n$, si vuole trovare la funzione f tale che

$$f(x_i) \approx y_i. \tag{2.4}$$

Per raggiungere questo scopo si suppone che la funzione f abbia dei parametri che devono essere calcolati e che, nel nostro caso specifico, sono rappresentati da a_k. Si vuole quindi trovare quei parametri a_k che minimizzino la somma dei quadrati dei residui (S):

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{2.5}$$

. Nella predizione lineare i residui sono modellati da e_n, si ottiene quindi il seguente errore quadratico totale E

$$E = \sum_{n} {e_n}^2 = \sum_{n} \left(s_n + \sum_{k=1}^{p} a_k s_{n-k} \right)^2 \tag{2.6}$$

A questo punto si deve minimizzare E prima di specificare l'intervallo della sommatoria, per far questo si specifica che

$$\frac{\delta E}{\delta a_i} = 0, \qquad i \leq i \leq p \tag{2.7}$$

Dalla (2.6) e dalla (2.7) si ottiene l'insieme di equazioni:

$$\sum_{k=1}^{p} a_k \sum_n s_{n-k} s_{n-i} = -\sum_n s_n s_{n-i}, \qquad 1 \leq i \leq p \tag{2.8}$$

Queste equazioni sono dette *equazioni normali*. Per ogni definizione del segnale s_n (2.8), si formi un sistema di p equazioni in p incognite che può essere risolto per i coefficienti del predittore $\{a_k, 1 \leq k \leq p\}$ che minimizza E in (2.6).

Definita una formulazione per la predizione lineare bisogna passare alla computazione dei parametri, o meglio dei coefficienti a_k, $1 \leq k \leq p$, che possono essere calcolati risolvendo un sistema di p equazioni in p incognite, come quelle in (A.2). Esistono diversi metodi per effettuare le computazioni necessarie, ad esempio tramite riduzione di Gauss o tramite il metodo dell'eliminazione [18]. Questi metodi non specifici della predizione lineare richiedono $p^3/3 + O(p^2)$ operazioni e p^2 locazioni di memorizzazione. É possibile ridurre in maniera rilevante la complessità temporale e spaziale della computazione dei parametri andando a guardare la forma particolare della matrice di autocorrelazione, per i dettagli si veda l'Appendice A.1.1. Il passo successivo è espandere la matrice di autocorrelazione ed applicare uno vari metodi ricorsivi presenti in letteratura, in particolare il metodo di Durbin, come spiegato nell'Appendice A.1.2.

Si noti che nell'ottenere la soluzione per un predittore di ordine p, si calcolano le soluzioni per tutti i predittori di ordine inferiore a p. É inoltre importante sottolineare come il calcolo delle soluzioni delle equazioni normali (A.2) non consista nella parte computazionalmente più pesante. Infatti la computazione dei coefficienti di autocorrelazione richiede pN operazioni, che dominano la computazione temporale nel caso frequente in cui valga $N \gg p$. Un ulteriore passo è possibile ed è la normalizzazione dei coefficienti di autocorrelazione, passo che viene descritto nell'Appendice A.1.3

Tornando all'ambiente del Grid Computing è interessante notare come tale tecnica possa essere applicata nel caso della predizione dell'affidabilità delle macchine di una rete. É infatti possibile vedere la serie di guasti tipici di una macchina come una serie temporale caratterizzata da una qualche funzione "segnale" a noi sconosciuta. Tramite questa visione è possibile allora utilizzare la teoria matematica sopra descritta per predirre il futuro comportamento del nostro segnale dei "guasti", cercando di intuire quale sarà il suo prossimo andamento. Per far questo basterà vedere i vari tempi di guasto della macchina come i punti misurati del segnale e determinati i parametri a_k predirre il guasto successivo.

Questo metodo presenta però alcuni problemi. Innanzitutto la predizione lineare prevede un campionamento del segnale elettrico ad intervalli uguali,

cosa che non vale nel caso nel campionamento del tempo di attività effettuato da un sensore posto su una macchina all'interno di una Grid. All'avvenire di un guasto il sensore infatti come tutto il sistema si fermerà e quindi le rilevazioni in questo caso non saranno caratterizzate da un campionamento ad intervalli uguali, ma vi saranno discontinuità dovute al tempo di riavvio della macchina. Il secondo problema è prettamente matematico: gli utilizzi di tale tecnica per gli scopi della predizione dell'affidabilità sono stati effettuati in letteratura solo in maniera sperimentale e non vi sono dimostrazioni teoriche dell'adattamento di questa tecnica a questo caso specifico. Nel Capitolo 4 si mostreranno quindi i risultati ottenuti applicando un'implementazione di questa tecnica per la predizione dei guasti all'interno di una Grid, e si evidenzierà nel dettaglio le osservazioni relative all'efficienza di questo metodo.

2.2 Predizione tramite distribuzioni Weibull ed Iperesponenziali

Nei più disparati campi scientifici il modo migliore per determinare il comportamento futuro di un'ambiente studiato è definire un modello matematico che lo rappresenti. Alcuni ricercatori hanno seguito questa via per studiare l'affidabilità delle macchine negli ambienti informatici distribuiti. Per rappresentare l'affidabilità dei calcolatori si possono usare differenti distribuzioni statistiche, le più usate sono l'esponenziale, la Pareto, la Weibull e l'Iperesponenziale. Determinare quale sia la migliore è possibile solo in via sperimentale, ma una volta effettuate queste sperimentazioni, è possibile ricavare degli strumenti che permettano di capire il comportamento futuro delle macchine.

Nell'articolo [9] gli autori hanno dimostrato tramite sperimentazioni che le distribuzioni Iperesponenziali e Weibull sono le più adatte a rappresentare l'affidabilità delle macchine in ambienti di calcolo distribuito. L'adattabilità di tali distribuzioni a simulare i comportamenti dei tempi di vita dei componenti di qualsiasi sistema non è però qualcosa di nuovo, infatti spesso nella teoria dei guasti queste vengono utilizzate per tali scopi. Gli autori hanno inoltre sviluppato dei predittori che studiano il comportamento di una macchina facendo uso appunto di questi modelli. Si va quindi ora a descrivere le due distribuzioni Weibull ed Iperesponenziale, e come si può ricavare previsioni da esse.

2.2.1 Distribuzione Weibull

La *distribuzione Weibull* è caratterizzata da una funzione di densità data da

$$f_w(x) = \alpha\beta^{-\alpha}x^{\alpha-1}e^{-(x/\beta)^\alpha} \tag{2.9}$$

mentre la sua funzione di distribuzione cumulativa è

$$F_w(x) = 1 - e^{-(x/\beta)^\alpha} \tag{2.10}$$

dove α è chiamato parametro di *shape*, e β è chiamato parametro di *scale*. La funzione di distribuzione condizionata per una Weibull è invece data da

$$F_{X|X>t(x)} = 1 - e^{[(t/\beta)^\alpha - (x/\beta)^\alpha]}, \tag{2.11}$$

che dipende principalmente da t e non solo dalla differenza $x-t$ quando $\alpha \neq 1$. Quando $0 < \alpha < 1$, la probabilità che la componente di un sistema sopravviva un'altra unita di tempo aumenta con la crescita di t. Viceversa per $\alpha > 1$ la probabilità di sopravvivenza descresce e con $\alpha = 1$ la distribuzione invece è senza memoria, quindi identica ad una esponenziale. Questa struttura permette alla Weibull di modellare la "sopravvivenza" di un gran numero di ambienti diversi al variare del suo parametro di *shape*.

Il vantaggio rispetto all'Iperesponenziale è che quest'ultima è soltanto capace di modellare un tempo di vita atteso crescente. Si può dimostrare quest'affermazione mostrando che la "funzione di rischio", che è essenzialmente il tasso di fallimento, è decrescente in funzione del tempo per ogni distribuzione Iperesponenziale.

Stabilita la distribuzione da utilizzare, date le misurazioni è utile determinare i parametri α e β al fine di poter poi costruire degli stimatori, predittori che rispettino questa distribuzione. Dato un campione di dati $\{x_1 \dots x_n\}$, ci sono diverse tecniche comuni per stimare i due parametri basandosi su un campione dei dati. Il metodo scelto ed utilizzato da John Brevik, ricercatore dell'Università della California in Santa Barbara, è basato sul principo della *massimo verosimiglianza*. Uno stimatore di massima verosimiglianza o *maximum likelihood estimator* (MLE) è calcolato per ogni insieme dei dati basandosi sulle assunzioni che ciascun punto campionato x_i è determinato da una variabile aleatoria X_i, e che ciascuna X_i sia indipendente e distribuita identicamente. Il metodo definisce una *funzione di verosimiglianza* L, dipendente dai parametri della distribuzione, come prodotto della funzione di densità valutata sui punti campionati. Nel nostro caso questa funzione sarà data da

$$L(\alpha, \beta) = \prod_i f(x_i) = \prod_i \alpha\beta^{-\alpha}x_i^{\alpha-1}e^{(x/\beta)^\alpha} \tag{2.12}$$

Massimizzare L equivale a massimizzare la probabilità congiunta che ciascuna variabile aleatoria prende sempre lo stesso valore del campione. Larghi valori della funzione di densità indicano che dati "più verosimili" sono stati prodotti. Per questi motivi nel caso dell'MLE bisogna semplicemente determinare i parametri α e β che massimizzano L. Al fine di semplificare il calcolo si effettua la massimizzazione non della funzione L ma del suo logaritmo, in quanto trattare una serie di somme è computazionalmente meno dispendioso che trattare una serie di prodotti. L'approccio numerico seguito dagli autori è quello di porre le derivate parziali di $logL$ uguali a 0 ed usare i risolvitori standard non lineari di equazioni al fine di trovare il punto critico corrispondente al massimo del logaritmo di L.

Esiste un altro metodo, qui non utilizzato, per la stima dei parametri conosciuto come *metodo dei momenti*. Questo metodo prevede di ricavare dei parametri per lo stimatore in maniera tale che il valore atteso e la varianza della distribuzione siano uguali al valore atteso campionario e alla varianza campionaria. A livello computazionale tale tecnica è più efficiente, ma il metodo dell'MLE presenta diverse proprietà e vantaggi statistici che hanno portato gli autori a scegliere quest'ultimo rispetto al metodo dei momenti.

2.2.2 Distribuzioni Iperesponenziali

Le distribuzioni Iperesponenziali sono essenzialmente una sommatoria pesata di distribuzioni esponenziali, ciascuna avente parametro differente. La funzione densita è data da:

$$f_H(x) = \sum_{i=1}^{k} p_i f_{E_i}(x) \tag{2.13}$$

dove

$$f_{E_i}(x) = \lambda_i e^{-\lambda_i x} \tag{2.14}$$

definisce la funzione di densità per una esponenziale avente parametri λ_i. Nella definizione di $f_H(x)$, per tutti i $\lambda_i \neq \lambda_j$ dove $i \neq j$ vale la seguente equazione

$$\sum_{i=1}^{k} p_i = 1 \tag{2.15}$$

La funzione di distribuzione cumulativa è definita invece come

$$F_H(x) = 1 - \sum_{i=1}^{k} p_i e^{-\lambda_i x} \tag{2.16}$$

per la stessa definizione di $f_{E_i}(x)$. Al fine di adattare l'Iperesponenziale ai dati forniti, il valore di k, di ciascuna λ_i, e di ciascuna p_i dev'essere stimato. Per

un valore specifico di k (il quale indica quante fasi devono essere incluse nell'Iperesponenziale), una tecnica MLE può essere utilizzata per determinare i rimanenti $2k - 1$ parametri.

Purtroppo questa operazione si rivela troppo spesso computazionalmente pesante e quindi gli autori hanno preferito utilizzare un altro algoritmo per la determinazione degli stimatori conosciuto come *estimation maximization* (EM) [7], che fornisce una soluzione computazionale buona ma non garantisce l'ottimalità.

2.2.3 Efficienza del metodo di previsione

L'efficienza di predittori tramite queste due distribuzioni Weibull è stata poi verificata utilizzandoli su tre data-set e si è mostrato come questi si comportino decisamente meglio dei precedenti stimatori, che solitamente utilizzavano distribuzioni di tipo *Pareto* o *Esponenziali*. Abbiamo deciso di verificare personalmente questi predittori implementandoli successivamente all'interno del nostro lavoro di tesi ed abbiamo riscontrato interessanti risultati che vengono descritti nel Capitolo 4.

2.3 Network Weather Service

Il *Network Weather Service* (NWS) è un sistema estensibile [46] per la generazione dinamica di previsioni sulle prestazioni delle risorse negli ambienti di calcolo distribuito. Questo sistema è pensato per essere il più modulare possibile, ed i due elementi principali che lo compongono sono dei *sensori* per il recupero di misurazioni del comportamento delle risorse e delle *previsioni* che sono calcolate a partire dalle misurazioni raccolte. In questa tesi non vengono considerati gli aspetti architetturali (perché fuori dallo scopo di questo studio) ma si illustra come questo strumento crei predizioni sul comportamento delle risorse [45].

2.3.1 Metodi di previsione in NWS

Il sistema NWS sfrutta un insieme di metodi di previsione che possono essere invocati in maniera dinamica, passando i parametri delle misurazioni ricavate da ogni risorsa. Dopo che ciascuna nuova misurazione viene raccolta, viene passata a tutti i metodi, ed una nuova previsione viene generata. Questo accade per ogni metodo di previsione f al tempo di misurazione t,

$$prediction_f(t) = METHOD_f(value(t), history_f(t)) \tag{2.17}$$

dove

- $value(t)$ corrisponde al valore misurato al tempo t
- $prediction_f(t)$ corrisponde al valore predetto dato dal metodo f per la misurazione $value(t+1)$,
- $history_f(t)$ corrisponde ad una storia finita delle misurazioni, previsioni e dei residui generata precedentemente al tempo t utilizzando il metodo f,
- $METHOD_F$ corrisponde al metodo di previsione f

I valori forniti dai vari sensori sono trattati come una serie temporale utilizzata dai metodi di previsione, e ciascun metodo mantiene una storia delle attività precedenti e delle informazioni sull'accuratezza delle previsioni. In particolare

$$err_f(t) = value(t) - prediction_f(t-1) \tag{2.18}$$

rappresenta l'errore residuo associato ad una misurazione e ad una predizione generata dal metodo f. L'attuale implementazione genera una previsione ogni volta che una nuova misurazione si rende disponibile. Dato che ogni metodo è valutato ogni volta che un nuovo dato è disponibile si sono scelti metodi dalla limitata complessità computazione, come descritto nella successiva sezione.

2.3.2 Metodi basati sulla media

NWS dispone di una classe di predittori che utilizzano metodi aritmetici che forniscono una previsione basata su una stima del valore medio sopra una determinata porzione della storia delle misurazioni. Una di queste è la *running average* definita come

$$RUN\_AVG(t) = \frac{1}{t+1}\sum_{i=0}^{t} value(i) \tag{2.19}$$

che utilizza la media di tutte le misurazioni raccolte al tempo t. Il valore medio calcolato precedentemente servirà come predittore per la misurazione che verrà raccolta al tempo $t+1$.

Questo tipo di predittore considera tutta la storia delle misurazioni ogni volta che genera una predizione, il peso computazionale generato da ciascuna di esse decresce linearmente nel tempo.

Se il valore più recente predice meglio la prossima misurazione, allora la media presa sopra una storia di lunghezza fissa fornirà una predizione

migliore. Per considerare questo fenomeno è stato quindi creato un predittore che utilizza "finestre mobili":

$$SW\_AVG(t,K) = \frac{1}{K+1} \sum_{i=t-K}^{t} value(i) \tag{2.20}$$

dove $K >= 0$ è un intero che rappresenta il numero di campioni da considerare nella finestra. Si noti che per $K = 0$, SW_AVG utilizza solo l'ultima misurazione come predittore, questo viene espresso dalla seguente:

$$LAST(t) = SW\_AVG(t,0) \tag{2.21}$$

La scelta a priori di K per SW_AVG può essere difficile da determinare per ciascuna risorsa e può variare nel tempo. Al fine di specificare dinamicamente K in maniera tale ad adattarsi alle serie temporali, si applica allora una strategia di "discesa del gradiente" implementata nel metodo *ADAPT_AVG(X)* descritto nell'Appendice A.3.1.

2.3.3 Metodi basati sulla mediana

Il valore mediano può funzionare anch'esso da predittore. In particolare se la sequenza delle misurazioni contiene valori occasionali che si discostano dalla media. Si può usare la mediana di una serie di misurazioni per prevedere quella al passo $t+1$. Per fare questo il valore mediano viene definito sopra una finestra mobile di lunghezza fissa in cui il limite destro è dato dalla misurazione più recente. Questo si definisce come

$$\begin{array}{lcl} Sort_K & = & \text{sequenza ordinata delle } K \text{ misurazioni più recenti} \\ Sort_{K(j)} & = & \text{il } j^{mo} \text{ valore nella sequenza ordinata} \\ MEDIAN(t,K) & = & \begin{cases} Sort_K((K+1)/2) & \text{se } K \text{ è dispari} \\ \frac{Sort_K(K/2)+Sort_K(K/2+1)}{2} & \text{se } K \text{ è pari} \end{cases} \end{array} \tag{2.22}$$

É possibile analogamente ai predittori basati sulla media la creazione di un filtro mediano adattativo che viene spiegato in dettaglio nell'Appendice A.3.2.

2.3.4 Modelli autoregressivi

Un altro metodo di previsione è stato realizzato tramite i modelli *Autoregressive integrated moving average* (ARIMA). L'adattamento di questi modelli ad una specifica serie temporale richiede la soluzione simultanea di un sistema di equazioni non-lineari, rendendo questa una procedura computazionalmente ardua in una configurazione dinamica.

L'adattamento di un modello puramente *autoregressivo* (AR) richiede invece solo la soluzione di un sistema lineare di equazioni che può essere risolto tramite ricorsione di Levinson [32]. La forma generale di un modello autoregressivo di ordine p è data da

$$AR(t,p) = \sum_{i=0}^{p} a_i * value(t-i) \tag{2.23}$$

Se la serie temporale è stazionaria, allora la sequenza $\{a_i\}$ che minimizza l'errore totale può essere determinato dalla soluzione del sistema lineare

$$\sum_{i=0}^{N} a_i * r_{i,j} = 0 \qquad j = 1, 2, \ldots, N \tag{2.24}$$

dove r_i, j è la funzione di autocorrelazione per le serie delle N misurazioni prese. I modelli autoregressivi risolvono un problema differente dai predittori lineari, ma numericamente sono equivalenti, si riporta quindi il lettore alla sezione relativa a questi per un approfondimento 2.1.

2.3.5 Selezione dinamica del predittore

Scegliere il corretto metodo predittivo per ciascun risorsa non è facile. Si possono però fare delle assunzioni relative alla risorsa in uso e stabilire ad esempio che per certi periodi di tempo conviene utilizzare un metodo e poi cambiarlo con uno che si adatti meglio nella nuova situazione. NWS al posto di cercare di determinare a priori un metodo corretto, decide di applicare contemporaneamente tutti i metodi di previsione. Poi per ciascun metodo guarda il residuo relativo al valore vero e produce una metrica per ciascuno di questi. Il metodo che dimostra di essersi comportato meglio per quel determinato tempo t viene utilizzato per generare la previsione al tempo $t+1$.

NWS usa l'errore quadratico medio

$$MSE_f(t) = \frac{1}{t+1} \sum_{i=0}^{t} (err_f(i))^2 \tag{2.25}$$

e l'errore medio percentuale

$$MPE_f(t) = \frac{1}{t+1} \sum_{i=0}^{t} |(err_f(i)|/value(i) \tag{2.26}$$

che serviranno per le metriche per ciascun metodo f al tempo t. Queste definite come

$$\begin{array}{l} MIN\_MSE(t) = predictor_f(t) \\ \text{se } MSE_f(t) \text{ è minimo rispetto tutti gli altri metodi al tempo } t \\ MIN\_MPE(t) = predictor_f(t) \\ \text{se } MPE_f(t) \text{ è minimo rispetto tutti gli altri metodi al tempo } t \end{array} \tag{2.27}$$

In tal maniera, al tempo t, il metodo presentate il più basso errore quadratico medio è utilizzato come previsione per la prossima misurazione da *MIN_MSE*. In maniera simile il metodo di previsione che al tempo t presenta il più basso errore percentuale totale diventa la previsione *MIN_MPE*.

NWS presenta quindi uno spettro piuttosto vario di metodi di previsione e tramite il sistema appena descritto, permette di scegliere la migliore per ogni caso, nel lavoro di tesi si è implementato un wrapper per richiamare questi metodi di previsione al fine di poterne testare l'efficacia, come descritto nel dettaglio nel Capitolo 3.

2.4 Grid Harvest Service

Il *Grid Harvest Service* (GHS) [47] è un valutatore di prestazioni ed un sistema per la schedulazione dei task e per la computazione di applicazioni su larga scala in ambiente condiviso. Il sistema è modulare e fornisce diversi servizi, al suo interno è presente un sistema elaborato che si basa sulla modelizzazione e la predizione delle prestazioni all'interno di una rete distribuita di calcolatori.

In questa trattazione non presenteremo l'architettura del sistema ma ci soffermeremo sul modello matematico che è stato realizzato [26]. Questo risulta interessante anche se tratta la predizione dell'affidabilità ad un livello differente dalle altre tecniche qui discusse. GHS tratta non solo il tempo in cui una macchina non subisce guasti ma anche il carico dovuto alla presenza o meno di task locali o remoti in esecuzionisul processore.

L'approccio di GHS mira a studiare e ad analizzare la natura del calcolo computazionale in maniera tale da sviluppare un approccio pratico per la stima e la predizione delle prestazioni, al fine di poter compiere scelte consapevoli per la distribuzione dei task.

2.4.1 Creazione ed analisi di un modello delle prestazioni

Prima di descrivere il modello bisogna fare alcune assunzioni. La prima assunzione riguarda il considerare che i task paralleli vengano assegnati soltanto alle macchine inattive o idle. La seconda assunzione è che il task parallelo sia composto da una singola fase di parallelismo iniziale, da qui si suddividano i sotto-task e questi siano senza legami e scambi di messaggi fino alla sincronizzazione finale. Altra assunzione è ritenere i tempi di ritardo dovuti alle comunicazioni tra le varie macchine inclusi nel tasso di servizio e considerare tutte le macchine dotate della stessa potenza computazionale, ossia facenti parte di un ambiente omogeneo.

Si distingue inoltre tra l'elaborazione dei *lavori locali* di tipo sequenziale eseguiti sulla macchina dal singolo proprietario e tra l'elaborazione dei *task paralleli* remoti. I primi avranno priorità sui secondi in termini di elaborazione computazionale. L'arrivo di un lavoro locale k si assume che segua una distribuzione di Poisson con tasso λ_k. Un lavoro sequenziale nuovo deve attendere se un altro lavoro sequenziale è in esecuzione.

Si consideri che il tempo di esecuzione dei lavori locali su una macchina k segua una distribuzione generale con media $1/\mu_k$ ed una deviazione standard σ_k. μ_k è anche chiamato il tasso di servizio sulla macchina k, in quanto questo dipende direttamente dalla potenza computazionale della stessa k.

Si assume che il task parallelo completo richieda un tempo di processamento pari a W ed che sia partizionato in m *sotto-task*, $w_1, w_2, \ldots, w_m$ per il processamento parallelo. Il sotto-task w_k è assegnato alla macchina k e si ha $W = \sum_{k=1}^{m} w_k$. Si usa T_k per rappresentare il tempo totale richiesto a finire il sotto-task k sulla macchina k. Nell'Appendice si mostra in dettaglio come si presentano il tempo di completamento di un sotto-task sulla singola macchina (A.2.1) ed il tempo di completamento di tutto il task parallelo (A.2.2).

2.4.2 Soluzione proposta da GHS

Basandosi sul modello analitico precedentemente esposto e dai risultati sperimentali, gli autori hanno proposto la seguente procedura utilizzata per determinare il miglior numero di calcolatori per il processamento parallelo in un ambiente eterogeneo:

1. Si inizia compilando una lista delle macchine che sono apparse poco cariche durante un periodo di tempo osservato.

2. Si determina il numero delle macchina da utilizzare, utilizzando l'equazione della strategia di partizionamento (A.36) per partizionare e allocare i sotto-task su ciascuna macchina.

3. Si predice poi la media e la deviazione standard del tempo di completamento parallelo tramite la (A.28) utilizzando la distribuzione Gamma per approssimare la variabile aleatoria $U(S_k)|S_k > 0$.

4. Si ripetono i passi 2 e 3 con un numero differente di macchine per identificare il numero migliore di queste che dev'essere usato in quel particolare ambiente distribuito.

Questa procedura si basa su due fasi: la prima riguarda la distribuzione del carico di lavoro tramite la (A.36), e la seconda si basa sulla predizione delle prestazioni tramite la (A.28). Dove queste due equazioni sono il risultato principale degli studi esposti dai creatori di GHS.

2.5 Predittori ibridi

Nella letteratura relativa alla predizione dei guasti spesso si adottano tecniche che mirano a selezionare il metodo più adatto alla storia passata delle misurazioni che si ha a disposizione. Come si è visto nella sezione 2.3.5 anche il Network Weather Service adotta una tecnica simile. Si è deciso quindi di creare un metodo di selezione nostro che determinasse tra i vari metodi a dispozione quale fosse il migliore data la storia passata della macchina. Siano dati N metodi di previsione e l'insieme delle m misurazioni totali rilevate sulla macchina *dataset*, con la grandezza di quest'ultimo $m \geq 20$, allora si procede in tal maniera:

1. si divide *dataset* in due insiemi uguali, al primo insieme si darà il nome di *trainingset*, al secondo insieme il nome di *testingset*;

2. si ottengono N valori predetti usando come misurazioni i valori all'interno di *trainingset*;

3. viene scelta una predizione alla volta e si calcola il residuo rispetto ad ogni misurazione contenuta nell'insieme *testingSet* e si calcola l'*errore residuo medio*;

4. si seleziona come "vincente" la previsione che ha ottenuto il *minimo errore residuo medio*;

5. si calcola la previsione finale utilizzando il metodo di previsione vincente sopra tutto il *dataset*.

Questo metodo di previsione è stato ispirato dalle tecniche classiche utilizzate in campi differenti quali il *machine learning*, i risultati sono stati interessanti, quindi si è intenzionati in futuro a lavorare ulteriormente attorno a questi metodi di selezione.

2.6 Conclusioni

Nella letteratura sono stati creati diversi metodi di previsione che data la storia o il comportamento di un elemento del sistema miravano a prevederne il prossimo valore. Le tecniche di previsione che sono state presentate nel capitolo appena concluso sono le più conosciute per quanto riguarda la previsione nell'ambito del Grid Computing. L'interesse verso di esse è mirato al loro utilizzo per il miglioramento dei problemi di allocazione dei lavori da parte degli scheduler. Questo aspetto dev'essere considerato attentamente insieme alle prestazione del metodo di previsione stesso, in quanto una buona previsione applicata male si dimostra inutile. Nell'ultimo capitolo di questa trattazione vengono mostrati i risultati ottenuti dalle nostre sperimentazioni e si mostra qual'è stato il comportamento di questi metodi secondo i dati da noi calcolati.

Capitolo 3

Implementazione dei metodi di previsione

Il lavoro di implementazione svolto per questa tesi si basa sugli argomenti trattati durante il laboratorio del corso di Sistemi Distribuiti. Tale lavoro consisteva nello sviluppo di politiche di scheduling knowledge-aware come estensione di un middleware per Grid chiamato OurGrid (descritto nel Capitolo 1). Il lavoro prevedeva lo studio di tre politiche basate sull'efficienza computazionale e sulla disponibilità delle macchine. All'interno di questo laboratorio si sono appresi i principali problemi legati all'allocazione dei task in ambiente Grid Computing e alla loro implementazione in sistemi software reali. Durante questo laboratorio l'interesse dei partecipanti e dei docenti si è sviluppato sui metodi di previsione legati all'affidabilità e così si è pensato di realizzare una libreria utilizzabile all'interno di OurGrid che permettesse di provare e sperimentare questi concetti teorici. In questo capitolo si descrive inizialmente il lavoro fatto durante il corso di sistemi distribuiti, poi in seguito si passa a parlare delle librerie sviluppate che mettono in pratica i metodi descritti nel Capitolo 2 ed infine si descrivono gli strumenti sviluppati per simulare e provare questi metodi.

3.1 Laboratorio di sistemi distribuiti

L'obiettivo del laboratorio era quello di prendere confidenza con le tecniche di schedulazione relative all'ambiente del Grid Computing introdotte durante il corso teorico. Il lavoro consisteva nell'utilizzo e nell'estensione del middleware OurGrid. Questo software utilizza delle tecniche di schedulazione predefinite, l'obiettivo era quello di estenderlo in maniera tale da poter utilizzare politiche diverse da quelle esistenti. Nella prima fase del laboratorio

si è preso confidenza con l'interfaccia di OurGrid e si sono condotti esperimenti di amministrazione di una Grid. Nella seconda fase di laboratorio si è studiato il codice sorgente di OurGrid e si sono evidenziate le prime problematiche. La terza fase consisteva nell'estensione ed implementazione delle proprie politiche di scheduling. Si inizierà a parlare dell'ambiente di lavoro su cui si sono condotte le sperimentazioni, e si proseguirà fino a parlare delle politiche di scheduling create.

3.1.1 Ambiente di lavoro

Al fine di poter effettuare e configurare una Grid tramite il sistema OurGrid il laboratorio era stato fornito di alcune macchine sopra le quali erano stati installati gli oggetti opportuni al funzionamento di tale sistema. Con l'intento di semplificare il processo di sviluppo e testing del codice, si è realizzata una piccola Grid locale sulla macchina usata per lo sviluppo, in maniera tale da non dover dipendere da quella scolastica. Si procede ora a dare una descrizione della Grid di prova utilizzata.

Descrizione della Grid di prova

La Grid di prova ha il nome di "thagrid" e consiste in:

- 1 Peer;
- 5 GuM

Inizialmente si era pensato di utilizzare delle macchine virtuali quali Xen, per creare l'ambiente di prova, purtroppo questo avrebbe rallentato ulteriormente la potenza di calcolo della macchina su cui si conduceva lo sviluppo e ci si è limitati a configurare opportunamente i vari User Agent e a farli puntare tutti all'interfaccia di rete locale 127.0.0.1.

Il Peer della Grid ha il compito di coordinare i vari useragent presenti in essa, il file descrittore che lo caratterizza è il seguente:

```
GuMdefaults:                                                    1
    site : mysite.com                                           2
    type : ualinux                                              3
    os : linux                                                  4
    processorfamily : IA32                                      5
    mem : 512                                                   6
    remExec : ssh -x $machine $command                          7
    copyFrom : scp $machine:$remotefile $localfile              8
    copyTo : scp $localfile $machine:$remotefile                9
GuM:                                                            10
    name: ua1.thagrid                                           11
    port: 3091                                                  12
    clockrate: 100                                              13
```

```
    cpuloadname: cpu1.txt
    uptimename: uptime1.txt
GuM:
    name: ua2.thagrid
    port: 3092
    clockrate: 300
    cpuloadname: cpu2.txt
    uptimename: uptime2.txt
GuM:
    name: ua3.thagrid
    port: 3093
    clockrate: 400
    cpuloadname: cpu3.txt
    uptimename: uptime3.txt
GuM:
    name: ua4.thagrid
    port: 3094
    clockrate: 200
    cpuloadname: cpu4.txt
    uptimename: uptime4.txt
GuM:
    name: ua5.thagrid
    port: 3095
    clockrate: 300
    bpinc: 0.3
    cpuloadname: cpu5.txt
    uptimename: uptime5.txt
```

All'interno di questo descrittore, ogni GuM è definito tramite tre parametri aggiuntivi che servono per gli scopi del progetto di laboratorio, questi tre parametri sono:

1. *clockrate*: contenente un numero intero positivo indicante la potenza di calcolo del processore;

2. *cpuloadname*: un file di testo diverso per ogni GuM, contenente le previsioni calcolate tramite NWS sul carico del processore;

3. *uptimename*: un file di testo diverso per ogni GuM, contenente le previsioni calcolate tramite NWS sull'affidabilità o tempo previsto di uptime della macchina.

Si è introdotto anche un parametro aggiuntivo *bpinc* che viene utilizzato per le macchine biprocessore, il cui utilizzo verrà spiegato in seguito. I due file di testo, sono collocati all'interno della directory *$STORAGE* di ogni GuM, nella Grid di prova queste directory sono:

```
/home/guido/.mgstorage1
/home/guido/.mgstorage2
/home/guido/.mgstorage3
/home/guido/.mgstorage4
/home/guido/.mgstorage5
```

Si noti che a meno di specificare diversamente tramite un path assoluto, le directory di storage sono contenute nella home directory dell'utente.

Network Weather Service

Il Network Weather Service è un sistema che periodicamente monitora e fornisce predizioni sulle prestazioni di una rete e sulle sue risorse computazionali. Questo strumento è esterno ad OurGrid e deve essere utilizzato per ricavare per ogni GuM della Grid di prova dei file di testo contenenti dati ed informazioni riguardanti:

1. il *carico di processore predetto* della macchina che ospitava lo User Agent;

2. il tempo di *uptime predetto* della macchina che ospitava lo User Agent.

Per generare la prima tipologia di file si è andati ad eseguire su quattro macchine del laboratorio il seguente comando:

```
nws_extract -h0 -f time,measurement -w availableCpu localhost |
 nws_add_forecast > cpuload.txt
```

questo comando si compone di due parti, tramite *nws_extract* si genera una sequenza di coppie di numeri con timestamp di rilevazione e carico corrente del processore:

```
# nws_extract -h0 -f time,measurement -w availableCpu localhost
1161273348 0.930880000
1161273358 0.939530000
1161273368 0.948350000
1161273378 0.957330000
```

e tramite il secondo comando *nws_add_forecast* prendendo in input le due colonne precedenti, si ottengono le predizioni statistiche:

```
1161273519 0.741330 0.741330 0.004609
1161273528 0.553100 0.553100 0.006321
1161273538 0.564620 0.564620 0.005996
```

Per l'estrazione delle informazioni riguardanti l'uptime si è scritto un breve codice che ha lo scopo di generare una coppia di valori timestamp, uptime corrente. Questo programma è stato lanciato su quattro macchine diverse, ma è stato catturato un solo valore di uptime equivalente al valore di uptime corrente. Questa scelta non è completa, in quanto si dovrebbe far eseguire lo script a diversi orari della giornata e per un periodo di tempo piuttosto lungo per catturare il reale tempo di uptime previsto. Ma per gli scopi del progetto questa semplificazione risultava tollerabile.

Questi file sono stati copiati nelle varie directory di *$STORAGE* definite per ogni GuM e sono serviti come contenitori dei dati di input rilevanti per la creazione di una schedulazione che sfrutti *uptime*, *cpuload* e *clockrate*.

3.1.2 Selezione del Task

L'attuale struttura di OurGrid permette di descrivere i lavori da sottomettere alla Grid tramite un file chiamato *Job Descriptor File* (JDF), questo oggetto però presenta alcune limitazioni. Tra quelle che più hanno influito sul progetto di laboratorio, vi è l'impossibilità di poter aggiungere dei parametri personalizzati relativi a ciascun task. Il testo d'esame per tale motivo richiedeva la costruzione di un programma che dato un elenco dei task ed un parametro predefinito dall'utente generasse un file JDF con gli stessi task ordinati in maniera crescente o decrescente. Si è deciso quindi di creare un nuovo *Extended Job Descriptor File* (EJDF) differente dal JDF originale di OurGrid, solo per l'aggiunta di un campo opzionale ai vari task, come nel seguente esempio:

```
task:                                                    1
        init:                                            2
        put ~/lib/nofwithdata.jar nofwithdata.jar        3
        extraparam: 40                                   4
        put ...                                          5
        remote : ...                                     6
        final : ...                                      7
```

L'idea è quella che l'utente possa scrivere il descrittore del suo job in maniera identica a come faceva originalmente, aggiungendo soltanto un ulteriore campo per ciascuno di essi. L'utente utilizza il programma passando questo JDF esteso come input, ed ottiene come risultato un file JDF classico che però presenta i task riordinati in maniera crescente o decrescente a seconda del valore intero associato ai vari parametri. Il programma EJDF richiamato senza argomenti restituisce una guida sintetica del suo utilizzo:

```
# java ejdf.Ejdf                                                          1
Usage: Ejdf [-c] [-d] [-output outputfile] [-param paramname] inputfile   2
```

L'unico argomento obbligatorio da inserire è *inputfile* cioè il nome del file EJDF da passare al programma, esso va a restituire in maniera predefinita un file JDF standard ordinandolo a seconda della presenza del campo *exetime*, nel caso i task non presentassero questo campo, si ignora l'ordinamento per il singolo task che viene aggiunto in testa al file.

Il programma EJDF presenta quattro parametri opzionali:

- *-c:* per definire che i task vengano ordinati in maniera crescente (di default);
- *-d:* per definire che i task vengano ordinati in maniera decrescente;
- *-output:* per indicare un file in cui scrivere l'output del programma;
- *-param:* per specificare il nome del parametro che si vuole usare per l'ordinamento e che sarà eliminato nell'output.

L'utilità di riordinare i task sul valore di un parametro viene evidenziata nella sezione 3.1.4, infatti questa soluzione permette di distinguere tra i task aventi un tempo di esecuzione corto e tra quelli aventi un tempo di esecuzione più lungo.

3.1.3 Lo scheduling su OurGrid

Prima delle nostre modifiche esistevano principalmente due politiche in OurGrid, delle quali quella predefinita è conosciuta sotto il nome di *Workqueue with Replication* (WQR). Questa politica di scheduling è tipo knowledge-free ossia non richiede nessun tipo di informazione per schedulare i task esistenti. Questo algoritmo ha dimostrato di avere prestazioni equivalenti ad algoritmi knowledge-aware. Descriviamo qui i moduli di OurGrid relativi al motore di scheduling esistente.

I package relativi allo scheduling sono i seguenti:

- org.ourgrid.mygrid.scheduler
- org.ourgrid.mygrid.scheduler.event
- org.ourgrid.mygrid.scheduler.exception
- org.ourgrid.mygrid.scheduler.gridmanager
- org.ourgrid.mygrid.scheduler.gump
- org.ourgrid.mygrid.scheduler.jobmanager

Gli oggetti più rilevanti per i nostri scopi sono inclusi principalmente nei gruppi *scheduler*, *gridmanager* e *jobmanager*. All'interno di questi package risiedono quattro oggetti che forniscono lo scheletro base per la costruzione dello scheduler, questo insieme di classi è stato gestito dagli sviluppatori di OurGrid come un sistema basato su eventi. Le quattro classi principali di questo modulo ad eventi sono:

- *EBSchedulerFacade* responsabile di mettere in coda gli eventi in SchedulerEventProcessor che poi andrà ad eseguire realmente le operazioni. Funziona quindi da punto di accesso per il modulo scheduler di OurGrid;
- *EBGridManager* si occupa del recupero dei GuM, utili alla schedulazione, andando a farne richiesta al GridMachineProvider, ossia facendone richiesta al Peer;

- *EBJobManager* è la classe responsabile della gestione dei job, essa mantiene una lista di tutti i lavori sottomessi dagli utenti e controlla tutte le operazioni legate ad essi;

- *EBSchedulingHeuristic* è un'interfaccia su cui vengono poi implementate tutte le euristiche di scheduling. Definisce quindi tutti i metodi principali che devono essere implementati. Ad esempio le due politiche di scheduling presenti Workqueue e Storage Affinity implementano questa interfaccia.

Questa gestione in prima analisi sembrava adatta all'estensione e creazione di nuove politiche, purtroppo andando ad effettuare prove pratiche e sperimentazioni, sono stati scoperti diversi problemi. Gli sviluppatori di OurGrid infatti hanno realizzato un sistema che sebbene modulare, è mirato ed orientato a fornire uno strumento efficiente per una schedulazione di tipo knowledge-free e non per una di tipo knowledge-aware. In dettaglio, per realizzare una schedulazione di tipo knowledge-aware su OurGrid è necessario inizialmente fissare un certo limite di tempo durante il quale si vanno a raccogliere le macchine disponibili sulla Grid, al fine di interrogare gli oggetti ad esse associate per ottenere appunto conoscenza delle loro caratteristiche (architettura, occupazione, statistiche etc). Un'altra possibilità è quella di attendere fin quando non vi è un certo numero di macchine disponibili e poi iniziare ad effettuare la schedulazione. Questo perché OurGrid non permette di ricevere gli oggetti relativi alle macchine, quindi i GuM, in maniera istantanea. Il software non permette questo tipo di gestione, in quanto la classe che si occupava di gestire gli eventi relativi alla schedulazione è strutturata in maniera inadatta, tale classe è la *SchedulerEventEngine*. Tale classe si occupa di chiamare il metodo *SchedulerEventEngine.schedule()* di ogni politica implementata, il problema nasceva dal fatto che essa inizialmente effettuava la chiamata al metodo precedentemente citato ogni volta che un evento veniva processato. Per i nostri scopi era necessario andare a processare un certo numero di eventi N e solo poi in seguito effettuare la chiamata allo scheduler. Parlando con gli sviluppatori e durante il corso, sono state proposte diverse soluzioni, quella che si è utilizzata per effettuare le sperimentazioni, risolve il problema precedente, ma ne introduce uno nuovo che ha come difetto l'utilizzo costante della CPU nel momento in cui si avvia il client MyGrid per sottomettere un job. Si mostra ora il codice originale della funzione *run()* del thread che gestiva la SchedulerEngine, spiegando i dettagli implementativi sui concetti presentati prima:

```
public void run() {                                    1
                                                       2
    boolean scheduled;                                 3
```

```
    while ( !mustShutdown ) {                                                4
                                                                             5
        scheduled = schedule( ebSchedulingHeuristic );                       6
                                                                             7
        ActionEvent actionEvent;                                             8
                                                                             9
        /*                                                                   10
         * If the schedule occurred successfully there could be more        11
         * replicas to be scheduled, so the schedule method must be called  12
         * even if there are no events in the event queue.                   13
         */                                                                  14
                                                                             15
        if ( scheduled ) {                                                   16
            actionEvent = eventQueue.unblockingRemove();                     17
        } else {                                                             18
            actionEvent = eventQueue.blockingRemove();                       19
        }                                                                    20
                                                                             21
        if ( actionEvent != null ) {                                         22
                                                                             23
            try {                                                            24
                actionEvent.process();                                       25
                                                                             26
            } catch ( Exception e ) {                                        27
                                                                             28
                LOG.error( "Error in event process! Cause: " +               29
                        e.getMessage(), e );                                 30
            }                                                                31
        }                                                                    32
    }                                                                        33
    this.isAlive = false;                                                    34
    shutdownEventQueue.put( new ShutdownResponseEvent() );                   35
}                                                                            36
```

Prima di commentare il codice si spiega in maniera dettagliata il ruolo delle due funzioni:

- eventQueue.unblockingRemove();
- eventQueue.blockingRemove();

La *unblockingRemove()* (vedi riga 17) rimuove il primo evento dalla coda restituendolo come valore di ritorno in maniera che possa poi essere processato, questa chiamata è non bloccante, in quanto non mette in attesa il thread:

```
public synchronized T unblockingRemove() {                                   1
                                                                             2
    if ( eventQueue.size() > 0 ) {                                           3
        return eventQueue.removeFirst();                                     4
    }                                                                        5
    return null;                                                             6
}                                                                            7
```

La *blockingRemove()* rimuove il primo evento dalla coda come la funzione precedente, ma se viene chiamata con la coda vuota, mette in attesa il thread, con il risultato di avere una chiamata bloccante:

```
public synchronized T blockingRemove() {
    try {
        while ( eventQueue.size() == 0 ) {
            wait();
        }
    } catch ( InterruptedException e ) {
        LOG.error( "", e );
    }
    return eventQueue.removeFirst();
}
```

Analizzando il codice della *SchedulerEventEngine.run()* si può notare come si proceda inizialmente a chiamare una schedulazione e se questa ha avuto successo allora viene bloccato il thread, altrimenti si vanno a processare in maniera non bloccante tutti i processi, e poi in seguito si chiamerà nuovamente lo scheduler. Per realizzare una politica di scheduling di tipo knowledge-free si è dovuto modificare la funzione descritta precedentemente in maniera opportuna, al fine di riservare un certo numero N di eventi prima di procedere alla schedulazione, la versione modificata è la seguente:

```
public void run() {
    boolean scheduled = false;
    int processedEvents = 0;
    int N = 5;
    while (!mustShutdown) {
        ActionEvent actionEvent;
        actionEvent = eventQueue.blockingRemove();
        if (actionEvent != null) {
            try {
                actionEvent.process();
                processedEvents++;
            } catch (Exception e) {
                LOG.error("Error in event process! Cause: "
                        + e.getMessage(), e);
            }
        }
        if(processedEvents == N || actionEvent == null) {
            scheduled = schedule( ebSchedulingHeuristic );
            processedEvents = 0;
        }
    this.isAlive = false;
    shutdownEventQueue.put(new ShutdownResponseEvent());
}
```

L'idea è quella di riservare sempre un certo numero N di macchine prima di mettersi ad effettuare lo scheduling, per fare questo si è utilizzata solo la funzione eventQueue.unblockingRemove() questo ha portato come conseguenza che il processore della macchina su cui gira MyGrid fosse sempre in utilizzo per tutto il tempo in cui si attende la terminazione di un Job. La soluzione

sopra descritta può essere modificata per non consumare inutilmente cicli di clock andando ad inserire un attesa di un certo numero di millisecondi (nella nostra sperimentazione con un valore di 100ms) questo però non assicura che vengano prese tutte le macchine disponibili, in quanto vi è un tempo di ritardo, in cui queste non possono essere raccolte. Non abbiamo avuto ancora modo di sperimentare adeguatamente questa soluzione su macchine diverse e scalando la grandezza della Grid.

I primi esperimenti per la creazione di una nuova politica di scheduling sono stati condotti analizzando la politica di scheduling standard contenuta all'interno della classe Workqueue. Questa classe è infatti un'estensione della *EBAbstractSchedulingHeuristic*, che è la classe astratta da cui bisogna poi far estendere tutte le altre politiche di schedulig. É importante citare il funzionamento in breve della politica Workqueue perché il lavoro di laboratorio seppur profondamente differente si è basato su di essa. La funzione *schedule()* di questa politica è identica a quella da noi usata, ed è la seguente:

```
public boolean schedule() {                                          1
                                                                     2
    List<TaskEntry> tasksToScheduleQueue = getTasksToScheduleQueue();  3
                                                                     4
    Iterator<TaskEntry> tasksIt = tasksToScheduleQueue.iterator();   5
    while ( tasksIt.hasNext() ) {                                    6
        TaskEntry task = tasksIt.next();                             7
        if ( task.canReplicate() ) {                                 8
            GuMClient chosenGuM = chooseGuM( task );                 9
            if ( chosenGuM != null ) {                               10
                ReplicaEntry replica =                               11
                        jobManager.createNewReplica( task );         12
                executeReplica( replica, chosenGuM );                13
                tasksIt.remove();                                    14
                return true;                                         15
            }                                                        16
        } else {                                                     17
            tasksIt.remove();                                        18
        }                                                            19
    }                                                                20
                                                                     21
    return false;                                                    22
}                                                                    23
```

Dal codice si osserva che prima vengono recuperati i task da processare e poi iterando per ognuno di essi si seleziona tramite la funzione *chooseGuM()* la macchina a cui assegnare il lavoro. La funzione chooseGuM() è la seguente:

```
private GuMClient chooseGuM( TaskEntry task ) {                      1
                                                                     2
    List<GuMClient> GuMs = gridManager.getAvailableGuMs(             3
                        task.getJobId(), task.getId() );             4
    if ( GuMs.isEmpty() )                                            5
        return null;                                                 6
    return GuMs.get( 0 );                                            7
}                                                                    8
```

Nella politica Workqueue si può vedere come data una lista delle macchine disponibili tramite la *gridManager.getAvailableGuMs* venga scelta arbitrariamente la prima libera per l'esecuzione del task. L'idea è stata quella di lasciare immutata la funzione di schedule() ed invece modificare la scelta della macchina, andando ad effettuare scelte e politiche dopo aver recuperato la lista delle macchine disponibili.

3.1.4 Creazione di politiche knowledge aware

Il testo d'esame prevedeva l'implementazione di tre politiche di schedulazione knowledge-aware su OurGrid. Le tre politiche utilizzate sfruttavano principalmente tre parametri che possono essere calcolati per ogni macchina che sono:

- *Clockrate:* il tasso in cicli al secondo (misurati in Mhz) tramite il quale l'elaboratore effettua le operazioni elementari. Nel nostro caso è indicato da un numero intero, inserito a mano nella definizione della Grid per ciascuna macchina;
- *Predicted uptime:* data una raccolta degli uptime attuali di ciascuna macchina, si vuole l'uptime predetto, calcolato tramite NWS;
- *Predicted CPU Load:* data una raccolta del carico del processore attuale di ciascuna macchina, si vuole il carico predetto calcolato tramite NWS;

Questi valori devono essere poi combinati per realizzare due politiche:

- *chooseGuMpredUptime:* questa prima politica sceglie la macchina più affidabile, cioè con maggiore uptime predetto;
- *chooseGuMeffCpu:* questa seconda politica sceglie la macchina più efficiente dal punto di vista della velocità di calcolo, ossia prendendo il miglior coefficiente *EffCpu*.

Dove *EffCpu* è

$$EffCpu = clockrate * cpuloadpredetto \tag{3.1}$$

Una terza politica invece doveva essere creata dal singolo studente, in modo tale da combinare in maniera utile i precedenti tre parametri, andando a sfruttare anche la selezione del task per ordine crescente o decrescente.

I metodi getPredUptime() e getEffCpu()

Dopo avere descritto quali sono i parametri che servono alle tre politiche in questione, si tratta brevemente come questi vengono recuperati. L'oggetto chiave per il recupero di informazioni relative ad una macchina in OurGrid è definito nella classe *GuMClient*, questa interfaccia ci permette di richiedere al Peer diverse informazioni relative al GuM selezionato, la classe *UserAgent* è implementazione di questa classe astratta. Per recuperare gli attributi descritti nel *Peer Descriptor File (PDF)* bisogna andare ad agire sulla classe *GuMSpec* che contiene appunto le specifiche definite dal gestore del Peer per ciascuna macchina, il recupero dei valori associati ai nomi degli attributi viene fornito tramite la *GuMSpec.getAttribute()*. Le funzioni rilevanti di GuMClient sono le seguenti:

- *getGuMSpecs():* per recuperare le specifiche associate ad ogni macchina e quindi gli attributi clockrate, cpuloadname, uptimename e bpinc;
- *getStorageDir():* per avere il percorso remoto della directory di storage contenente i file generati da NWS;
- *getFile():* per inviare al client MyGrid i file presenti sui vari GuM tramite le funzionalità offerte dalla *Java Remote Method Invocation* (RMI).

Tramite queste tre funzioni è possibile ricevere dal Peer tutto il necessario per avere le informazioni relative alle caratteristiche delle macchine a disposizione. Come vedremo, per ogni macchina il client MyGrid (descritto nel Capitolo 1) necessita di effettuare una chiamata al Peer per farsi inviare il file generato da NWS, questa soluzione era quella di più facile realizzazione ma ha posto diversi dubbi in quanto, nel caso di griglie molto estese, può causare problemi di scalabilità dovuti al trasferimento di pacchetti per ogni macchina. Servivano quindi due funzioni che recuperassero i parametri su cui sono basate le tre politiche create, questi due metodi sono *getPredUptime()* e *getEffCpu()*. Entrambe le funzioni sono molto simili ed il loro funzionamento si può riassumere in tre passi:

1. data una struttura *GuMClient* si recuperano i valori associati ai seguenti attributi:
    - *uptimename:* stringa rappresentante il nome del file contenente l'uptime predetto, recuperato nella getPredUptime().
    - *clockrate:* valore intero indicante il clockrate in Mhz della macchina esaminata, recuperato nella getEffCpu().

- *cpuloadname:* stringa rappresentante il nome del file contenente il cpuload predetto, recuperato nella getEffCpu().

2. Tramite la *getStorageDir* e la *getFile* si richiede l'invio remoto dei file il cui percorso è specificato negli attributi *uptimename* e *cpuloadname*. Ricevuti questi due file, si vanno a memorizzare nelle directory temporanee di sistema, ed analizzando l'ultima linea di ciascun file si recuperano i valori predetti.

3. Recuperati i valori precedenti si provvede a fornirli come valore di ritorno della funzione, nel caso di errori I/O o nel caso di sintassi errata, o di parametri mancanti, il valore ritorno è pari a zero e si registrano i malfunzionamenti nel file di log di MyGrid.

La scelta di fornire un valore di ritorno pari a zero è utile per non bloccare la schedulazione anche nel caso di parametri mancanti. Nel caso ipotetico di un Peer Descriptor File che non presenti nessuno dei tre attributi sopra specificati, le tre politiche si riducono essenzialmente ad una politica Workqueue, permettendo comunque l'esecuzione dei job. Al fine di poter permettere delle Grid che possano avere macchine monoprocessore o biprocessore è stato introdotto un ulteriore parametro da inserire nel Grid File Descriptor, questo parametro è *bpinc*. La presenza di tale parametro specifica alla funzione *getEffCpu* che la macchina è biprocessore e che quindi il valore raccolto dal file indicato da *cpuloadname* dev'essere normalizzato per due, mentre il valore del *clockrate* dev'essere incrementato del valore percentuale *bpinc* sul clockrate. Questo valore *bpinc* scelto dall'utente, deve stare tra 0 ed 1 e rappresenta l'incremento della macchina biprocessore al clockrate rispetto al suo equivalente monoprocessore. Questo valore è definito dall'utente e può essere ricavato tramite benchmark o indicato dall'amministratore della Grid.

I metodi chooseGuMpredUptime() e chooseGuMeffCpu()

Le due politiche di base richieste devono andare semplicemente a recuperare i valori predetti (*uptime* o *effCpu*) per ogni GuM disponibile, dati questi, si seleziona la macchina che possiede il valore più alto. In queste due politiche ed anche nella terza il metodo *schedule()* è lo stesso presentato precedentemente per la Workqueue, cambia invece il contenuto della funzione chooseGuM in modo da effettuare un incapsulamento della funzione desiderata:

```
private GuMClient chooseGuM(TaskEntry task) {                    1
    return chooseGuMpredUptime(task);                            2
}                                                                3
```

In tal maniera viene automaticamente scelto il GuM presentante la migliore caratteristica scelta, avremo quindi due politiche corrispondenti ai due metodi:

1. *chooseGuMpredUptime():* sceglie la macchina più affidabile, cioè con il tempo di uptime predetto più alto, andando a selezionare la macchina che tramite *getPredUptime()* ha avuto valore più alto;

2. *chooseGuMeffCpu():* sceglie la macchina più efficiente, cioè con il prodotto $clockrate * cpuload$ predetto più alto, andando a selezionare la macchina che tramite *getEffCpu()* ha avuto valore più alto.

La *chooseGuMpredUptime()* ad ogni schedulazione cerca la macchina che ha avuto il maggior tempo di uptime predetto, cioè la macchina più affidabile. I vantaggi di tale politica si presentano quando i task hanno un tempo di esecuzione "importante" anche indipendentemente dalla potenza di calcolo delle macchine presenti sulla Grid, in tal maniera si ha una certa garanzia che il task sottomesso abbia il tempo di poter finire indipendentemente dalle repliche.

La mia politica di scheduling: MyScheduling

Per realizzare la mia politica di scheduling ho cercato in letteratura soluzioni adoperanti come parametri quelli richiesti per l'esame. Non sono riuscito a trovare molte politiche knowledge-aware che scelgano la "macchina migliore" senza andare a rapportare questa al singolo task. Le politiche applicate di solito in ambito Grid scelgono la macchina migliore per un task e non globalmente. Attuare una politica simile con OurGrid non è possibile a causa dell'impossibilità di andare a specificare singoli attributi per i vari task. Il problema diviene quindi trovare una politica di scheduling che selezioni la macchina migliore sapendo solo che i task vengono poi selezionati in maniera crescente o decrescente. I parametri a disposizione sono l'efficienza di calcolo e l'uptime, questi due fattori possono avere rilevanza diversa a seconda del tipo di job che viene sottomesso. I parametri possono anche avere un interesse diverso a seconda dell'utente. Pensando a questi problemi, si è realizzata una politica che potesse dare la possibilità all'amministratore della Grid di determinare quanto fossero rilevanti i due parametri di decisione che si hanno a disposizione. Nella nostra politica, definita principalmente tramite la funzione *myChooseGum()*, è possibile variare l'importanza che si vuole dare a tali parametri. Questo impone però una certa attenzione da parte dell'amministratore della Grid che deve scegliere opportunamente tali valori. L'idea

è di selezionare date n macchine disponibili al momento della schedulazione la macchina i che presenti il migliore fattore E_i:

$$E_i = \alpha \frac{EffCpu}{\sum_{j=1}^{n} EffCpu_j} + \beta \frac{Uptime_i}{\sum_{j=1}^{n} Uptime_j} \qquad \text{con } \alpha + \beta = 1 \qquad (3.2)$$

dove α e β sono i "pesi" che specificano l'importanza attribuita all'efficienza del processore oppure attribuita alla stabilità della macchina. Come si può vedere vi è anche una normalizzazione sui valori totali dei due fattori in modo da poter poi successivamente pesare in maniera corretta tramite i valori α e β. É possibile modificare i parametri α e β andando a specificare i valori all'interno dei codici sorgenti della classe MySchedule. Durante gli esperimenti condotti questi due valori sono stati fissati a 0,8 e 0,2, in quanto i job che venivano sottoposti possedevano dei tempi di esecuzione molto più bassi dei valori di uptime che avevo inserito. Si può fissare questi valori a 0,5 e 0,5 nel caso si voglia dare la stessa importanza ad efficienza ed affidabilità, come si può fissarli a 0,2 e 0,8 nel caso di grossi task con tempi di esecuzione molto lunghi.

3.1.5 Riflessioni sul lavoro di laboratorio

I problemi nell'estensione di un motore di scheduling presenti in un software già esistente sono stati diversi, l'architettura di OurGrid è stata creata con determinati obiettivi, e spesso questi imponevano scelte ortogonali a quelle che sarebbero state ottimali per i nostri scopi. L'esperienza acquisita con il middleware di OurGrid e con il linguaggio Java hanno spinto ad approfondire il campo della predizione delle prestazioni ed usare Java come linguaggio principale per le sperimentazioni, come si mostra nella Sezione 3.2.

3.2 Predittori dell'affidabilità

L'aver fatto esperienza diretta sui problemi implementativi che occorrono durante la gestione dello scheduling in ambito Desktop Grid ha permesso di avere una visione su quali fossero le reali problematiche. In questa sezione si descriverà il lavoro successivo che è stato quello di realizzare una serie di predittori per l'affidabilità e riunirli all'interno di una libreria Java che ne rendesse facile l'integrazione con OurGrid o con qualsiasi altro strumento basato su questo linguaggio. L'implementazione e creazione di questa libreria è stata preceduta anche da un periodo consistente di ricerca al fine di scegliere i metodi e le tecniche più importanti presenti al momento nella letteratura accademica sull'argomento (come descritto nel precedente capitolo). Si inizia

a descrivere quali sono stati i passi iniziali di progettazione e poi si passa a mostrare la creazione di ciascun predittore.

3.2.1 Progettazione ed analisi del problema

La progettazione di una libreria contenente diversi metodi di previsione è stata caratterizzata da una fase iniziale, di definizione del problema, di analisi dei requisiti, di studio dei casi d'utilizzo, di prime implementazioni e continue revisioni che durano tutt'ora. Attualmente i predittori implementati sono i seguenti quattro:

- *NwsPredictor:* che fornisce classi e metodi per generare previsioni tramite le tecniche del Network Weather Service;
- *LinearPredictor:* che fornisce classi e metodi per generare previsioni tramite la tecnica della predizione lineare;
- *BrevikPredictor:* che fornisce classi e metodi per generare previsioni tramite i modelli matematici usati da Wolski e Brevik;
- *HybridPredictor:* che fornisce classi e metodi per generare previsioni selezionando tra quelle precedenti la migliore a seconda del tipo di campione di dati.

Si voleva realizzare un predittore per l'affidabilità/disponibilità delle macchine di un ambiente distribuito Grid, permettendo l'utilizzo di diverse metodologie proposte dalla ricerca in questo ambito. Al fine di soddisfare questo intento ci si è posti una serie di requisiti da rispettare:

- utilizzo del linguaggio Java e di una programmazione di tipo Object Oriented;
- piena integrazione con il middleware OurGrid;
- possibilità di integrare librerie e software esterni quali ad esempio le librerie di NWS o altri strumenti scritti in linguaggi diversi da Java;
- creare una libreria caratterizzata dalla massima generalità e portabilità.

L'ostacolo iniziale è stato pensare "ad oggetti" come realizzare una libreria che fosse il maggiormente estendibile e generica. Si sapeva che le sperimentazioni si sarebbero condotte su OurGrid e che quindi il linguaggio di riferimento doveva essere Java. Inoltre si dovevano utilizzare anche librerie esterne in altri linguaggi come per NWS. Inizialmente si è pensato quindi al *package* e a

come integrare il codice del primo predittore con OurGrid, all'inizio si aveva intenzione di creare un *sotto-package* interno a quello di OurGrid, ma si poi è giunti alla conclusione che era meglio separare il codice creando un package indipendente chiamato *unipmn.it.predictor* e reso totalmente indipendente da OurGrid. Lo strumento di previsione doveva permettere all'utente che

Figura 3.1: Casi di utilizzo di un predittore

sta sviluppando una politica di scheduling di

1. fornire le misurazioni ricavate da un qualche sensore in un determinato formato;
2. scegliere un metodo di previsione, quindi NWS, GHS, Weibull, etc.;
3. ricevere delle predizioni ottenute dalla storia passata della macchina in un determinato formato.

Sono stati stesi alcuni schemi cartacei ed UML (Figura 3.1) e si è fatto uso di strumenti per mettere velocemente in formato digitale le idee che sopravvenivano durante la progettazione (Figura 3.2). Si mostrano ora il package e le interfacce che forniscono lo scheletro della libreria per la predizione dell'affidabilità.

3.2.2 Package presenti nel predittore

L'implementazione attuale presenta sei package Java riguardanti le interfacce, i quattro predittori creati ed alcune classi usate per le sperimentazioni e le simulazioni:

- it.unipmn.dcs.predictor.common
- it.unipmn.dcs.predictor.nws

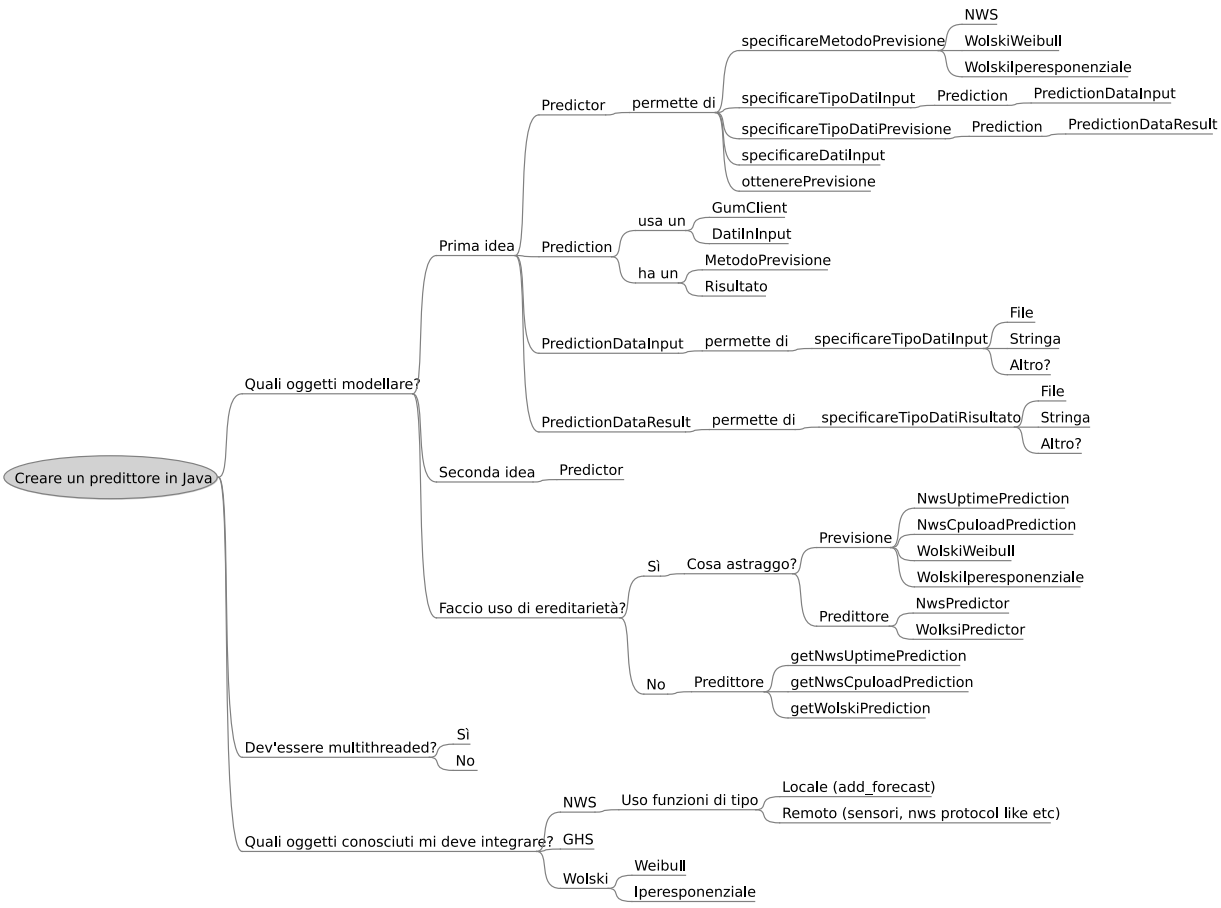

Figura 3.2: Schema mentale per la realizzazione di un predittore.

- it.unipmn.dcs.predictor.linear
- it.unipmn.dcs.predictor.brevik
- it.unipmn.dcs.predictor.hybrid
- it.unipmn.dcs.predictor.testing

Il primo package è relativo alle classi comuni e alle interfacce, i seguenti quattro ai vari predittori e l'ultimo agli strumenti di simulazione e sperimentazione.

3.2.3 Descrizione delle interfacce

Si descrive ora il codice relativo alle interfacce, innanzitutto mostrandone uno schema UML in Figura 3.3. La prima interfaccia denominata *Measurement*

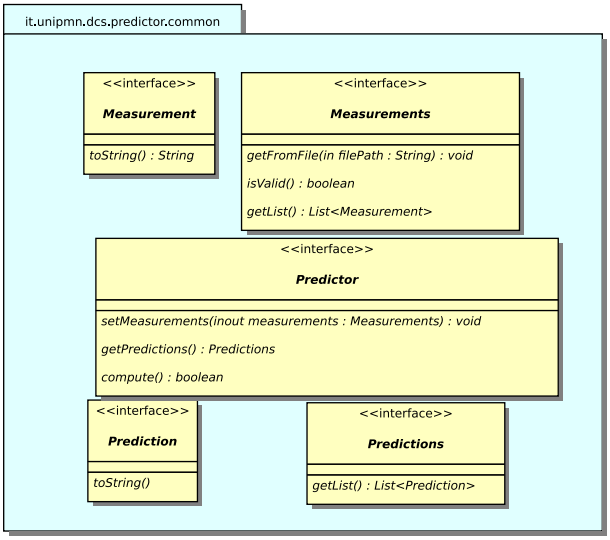

Figura 3.3: Schema delle interfacce

serve ad indicare i metodi obbligatori per le classi che conterranno una singola misurazione raccolta e presenta questo schema:

```
package it.unipmn.dcs.predictor.common;

public interface Measurement {

    public abstract String toString();

}
```

per ciascun oggetto che usi Measurement è stato pensato un contenitore che implementi i metodi di *Measurements*:

```
public interface Measurements {

    public abstract void getFromFile(String filePath) throws IOException;

    public abstract boolean isValid();

    public abstract List <Measurement> getList();
}
```

In Java le interfacce servono a definire uno schema principale dei metodi obbligatori per una serie di tipologie di oggetti. Com'è possibile notare i metodi obbligatori sono molto pochi e si limitano allo stretto necessario. La prima interfaccia è relativa alla singola misurazione, e presenta solo un metodo che restituisce una stringa con la misurazione già formattata. La seconda classe serve da contenitore per le varie misurazioni, e prevede i metodi per raccogliere i dati da un file, per validarli e per ricevere una lista di questi. Discorso analogo per le predizioni, con l'interfaccia *Prediction*:

```
package it.unipmn.dcs.predictor.common;

public interface Prediction {

    public abstract String toString();

}
```

e l'interfaccia *Predictions*:

```
package it.unipmn.dcs.predictor.common;

import java.util.List;

public interface Predictions {

    public abstract List<Prediction> getList();

}
```

Qui la struttura è più semplice in quanto non si vuole indicare una linea guida obbligatoria per la memorizzazione delle varie predizioni lasciando questa scelta al caso specifico del singolo predittore. L'ultima interfaccia è quella relativa al "motore" del predittore e alla sua politica, il cui codice è il seguente:

```
package it.unipmn.dcs.predictor.common;

public interface Predictor {

    public abstract void setMeasurements(Measurements measurements);

    public abstract Predictions getPredictions();

    public abstract boolean compute();

```

```
    public abstract boolean compute(int n);                    11
                                                               12
}                                                              13
```

I metodi astratti obbligano la presenza in ciascun predittore di un metodo *setMeasurements()* per il passaggio delle misurazioni, di un metodo *compute()* o *compute(int n)* per la computazione ed il calcolo delle previsioni o delle ultime n previsioni ed infine di un metodo *getPredictions()* per l'ottenimento delle previsioni.

3.2.4 Predittore NWS

Il predittore NWS sfrutta le tecniche descritte nella Sezione 2.3 per fornire previsioni sull'affidabilità di una macchina. Gli strumenti forniti da NWS sono distribuiti insieme al loro codice sorgente scritto in linguaggio C/C++. Esiste anche una libreria Java che però permette di accedere remotamente ad NWS ma di non sfruttare le sue tecniche di previsione. Dopo esserci consultati con uno dei programmatori di NWS ci è stato detto che l'unica maniera era quella di includere il codice C delle librerie di NWS all'interno del nostro codice Java tramite le librerie *Java Native Interface* (JNI) [33]. Le JNI sono un framework di programmazione che permette a del codice Java in esecuzione sulla *Java Virtual Machine* (VM) di chiamare ed essere chiamato da applicazioni native (programmi specifici ad un hardware e ad un determinato sistema operativo) e da librerie scritti in altri linguaggi quali C, C++ ed assembly. Si è fatto uso all'interno del predittore NWS delle JNI per accedere direttamente alle librerie native di NWS, questo inizialmente ha posto qualche difficoltà in quanto queste librerie non sono di facile utilizzo. Si è deciso di separare concettualmente il codice Java dalle librerie native creando una classe che effettuasse da livello intermedio tra i due, da "wrapper". Mostriamo innanzitutto lo schema UML del predittore NWS che si concretizza nel package *it.unipmn.predictor.nws* e in it.unipmn.predictor.nws.wrapper in Figura 3.4. Questo schema evidenzia i rapporti principali tra l'implementazione del predittore e le varie relazioni con le apposite interfacce. Come già spiegato il predittore è scritto in Java, ma sfrutta delle librerie già esistenti di NWS, nel nostro caso le librerie incluse nella versione 2.13. L'importante per un corretto funzionamento è che sul sistema in uso vi sia installata una versione di NWS, e quindi vi sia l'header file *nws_api.h* nei vari path di sistema (*/usr/include* o */usr/local/include*). Questo affinché l'utente possa, dati i sorgenti, compilare il predittore con successo. La classe *NwsWrapper* serve da strato di mezzo tra il codice Java e le librerie C ed include un metodo *addForecast*, che va a richiamare tramite le JNI le librerie native che effettuano il calcolo delle predizioni. La porzione di codice C ed il wrapper vengono

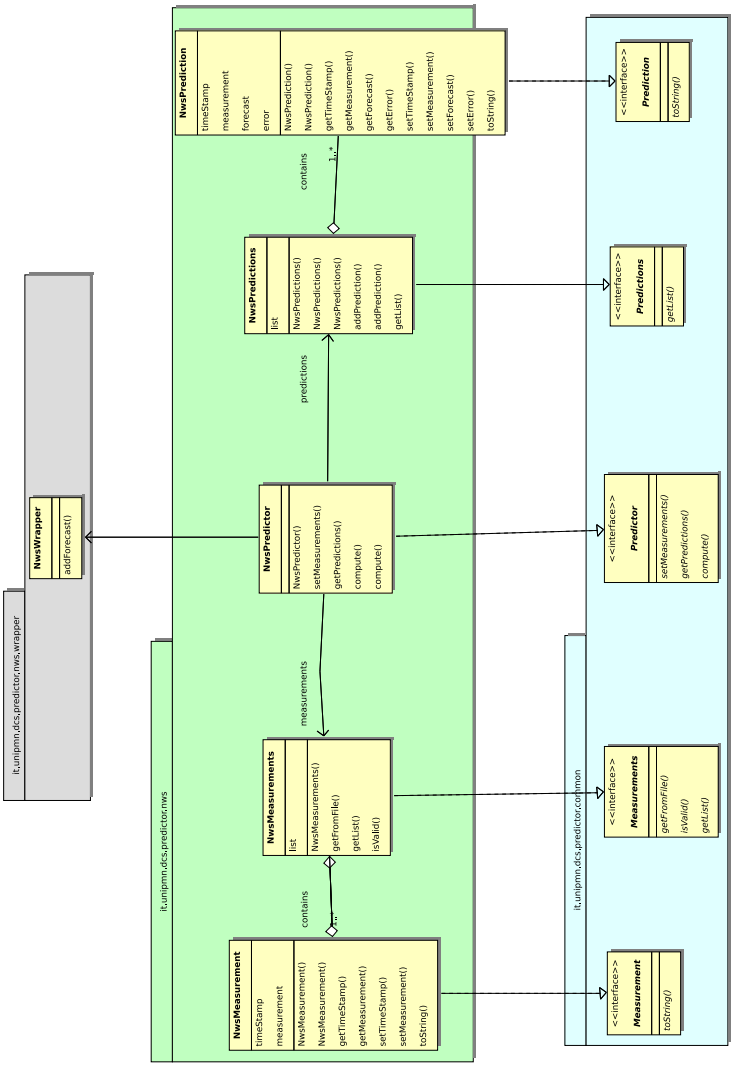

Figura 3.4: Diagramma UML per l'NwsPredictor

spiegati successivamente. Si descrivono ora in maniera approfondita i vari oggetti mostrando prima gli oggetti legati alle misurazioni e alle predizioni.

Le classi *NwsMeasurement* e *NwsMeasurements* implementano i metodi relativi alla memorizzazione dei valori numerici relativi alle misurazioni ed i metodi relativi all'ottenimento di questi da diversi input (file etc.). Si mostra il codice relativo a NwsMeasurement:

```
public class NwsMeasurement implements Measurement {                    1
                                                                         2
    private double timeStamp;                                            3
    private double measurement;                                          4
                                                                         5
    public NwsMeasurement(double timeStamp, double measurement) {..}     6
    public double getTimeStamp() {...}                                   7
    public double getMeasurement() {...}                                 8
    public void setTimeStamp(double timeStamp) {...}                     9
    public void setMeasurement(double measurement) {...}                 10
                                                                         11
    public String toString() {...}                                       12
```

Quest'oggetto contiene le informazioni relative ad una singola misurazione, questa è identificata dall'intervallo temporale *timeStamp* in cui è stata raccolta, e dal valore numerico della misurazione stessa *measurement*. Sono presenti vari metodi "getters" e "setters" ed il metodo *toString* che stampa i valori di timeStamp e measurement formattati secondo la codifica usata dagli sviluppatori di NWS. Si mostra ora il codice di *NwsMeasurements* che si occupa di tenere memorizzata una lista delle misurazioni (cioè dei singoli oggetti NwsMeasurement) e dei metodi per il recupero di queste da un file.

```
public class NwsMeasurements implements Measurements {                  1
                                                                         2
    private List <Measurement> list;                                     3
                                                                         4
    public NwsMeasurements(String filePath) throws IOException,          5
     NumberFormatException {
        getFromFile(filePath);                                           6
    }                                                                    7
                                                                         8
    public void getFromFile(String filePath) throws IOException {        9
                                                                         10
        File f = new File(filePath);                                     11
        .......                                                          12
                                                                         13
        while(line != null)                                              14
        {                                                                15
            ....                                                         16
                NwsMeasurement measurement = new NwsMeasurement(Double.  17
                 parseDouble(lm[0]), Double.parseDouble(lm[1]));
                this.list.add(measurement);                              18
        }                                                                19
                                                                         20
    }                                                                    21
                                                                         22
    public List <Measurement> getList() {...}                            23
    public boolean isValid() {...}                                       24
```

L'oggetto *NwsMeasurements* presenta un costruttore che preso il nome del file contenente le misurazioni, richiama la funzione *getFromFile()* che si occupa di leggere il file delle misurazioni, ed inserire i valori numerici di ciascuna di essa dentro un oggetto NwsMeasurement che viene poi aggiunto in una lista linkata. Vi è un poi un metodo per l'ottenimento dell'intera lista, ed uno per validarla *isValid()*. Quest'ultimo metodo però attualmente si occupa solo di controllare che la lista non sia nulla, in quanto il metodo *getFromFile* nel caso l'input del file non fosse ben formattato lancerebbe un eccezione *NumberFormatException*.

Le classi *NwsPrediction* e *NwsPredictions* implementano i metodi addetti alla memorizzazione dei valori numerici relativi alle predizioni e all'ottenimento di questi da diversi input (file etc.). L'oggetto NwsPrediction è analogo all'oggetto NwsMeasurement presentato, cambiano semplicemente i valori memorizzati:

```
public class NwsPrediction implements Prediction {                    1
                                                                       2
    private double timeStamp;                                          3
    private double measurement;                                        4
    private double forecast;                                           5
    private double error;                                              6
    ...                                                                7
```

Tramite quest'oggetto infatti memorizziamo quattro tipi *double* che contengono il *timeStamp* ed il valore numerico dell'ultima misurazione *measurement*, il valore predetto *forecast*, e l'errore relativo *error* a questo valore. L'oggetto *NwsPredictions* contiene invece la lista delle diverse predizioni, e dei metodi per l'inserimento delle singole:

```
public class NwsPredictions implements Predictions {                  1
                                                                       2
    private List <Prediction> list;                                    3
                                                                       4
    public NwsPredictions() {}                                         5
    public NwsPredictions(NwsPrediction prediction) {...}              6
    public NwsPredictions() {..}                                       7
    public NwsPredictions(String filePath) throws IOException{..    }  8
                                                                       9
                                                                       10
    public void getFromFile(String filePath) throws IOException,       11
     NumberFormatException {...}
                                                                       12
    public void addPrediction(NwsPrediction prediction) {...}         13
    public void addPrediction(double timeStamp, double measurement, double  14
     forecast, double error) {...}
                                                                       15
    public List <Prediction> getList() {    ...}                      16
```

Questo oggetto è analogo al NwsMeasurements, contiene una lista linkata per la memorizzazione degli oggetti NwsPrediction ed alcuni metodi *addPrediction()* che permettono di aggiungere i valori in maniera incrementale.

La classe *NwsPredictor* si occupa di prendere un insieme di misurazioni

definito da NwsMeasurements, effettuare il calcolo delle previsioni tramite un metodo *compute()* ed offrire l'insieme delle predizioni calcolate all'interno di NwsPredictions.

```
public class NwsPredictor implements Predictor {                          1
                                                                           2
    private NwsMeasurements measurements;                                 3
    private NwsPredictions  predictions;                                   4
                                                                           5
    public NwsPredictor (Measurements measurements, Predictions predictions) 6
    {...}                                                                  7
                                                                           8
    public void setMeasurements(Measurements measurements) {...}          9
    public Predictions getPredictions() {...}                              10
                                                                           11
    public boolean compute(int nLastPredictions) {                         12
        .....                                                              13
                                                                           14
            NwsWrapper nwsWrapper = new NwsWrapper();                      15
            nwsWrapper.addForecast(measurementsArray, predictions,         16
             nLastPredictions);
                                                                           17
        .....                                                              18
    }                                                                      19
                                                                           20
    public boolean compute() {                                             21
        return this.compute(20);                                           22
    }                                                                      23
```

Sono presenti un costruttore per il passaggio delle misurazioni e delle predizioni, ed i relativi setter e getters. Il metodo più importante è *compute(int nLastPredictions)* che va a richiamare la funzione *NwsWrapper.addForecast()* per effettuare il calcolo della previsione. Lo stesso metodo può essere anche utilizzato senza parametri, ed in questo caso calcola le ultime 20 previsioni. Questo numero è inserito dentro il codice e serve ad evitare nel caso di numerose misurazioni l'occupazione di troppe risorse di calcolo e memorizzazione. É importante inoltre notare che agli scopi di uno scheduler la previsione rilevante è soltanto l'ultima. Andiamo a mostrare ora il wrapper *NwsWrapper* dentro il package *it.unipmn.dcs.predictor.nws.wrapper*:

```
public class NwsWrapper {                                                  1
                                                                           2
    // Loading /usr/lib/libNwsWrapper.so                                   3
    static {                                                               4
        try {                                                              5
            System.loadLibrary("NwsWrapper");                              6
        ...........                                                        7
        ...........                                                        8
    }                                                                      9
                                                                           10
    public native void addForecast(                                        11
            Measurement [] measurements,                                   12
            Predictions predictions,                                       13
            int nLastPredictions);                                         14
}                                                                          15
```

Questa classe all'inizio avvisa il sistema che sarà necessario caricare una shared library scritta in C chiamata *libNwsWrapper.so*, questa libreria conterrà dei metodi nativi che verranno chiamati all'interno del codice Java. La funzione *addForecast* è definita come *public native void*, questo significa che andrà a chiamare un metodo C omonimo passandogli:

- un array di oggetti NwsMeasurement contenente le misurazioni lette da un file measurements[];
- un oggetto NwsPredictions che dovrà essere riempito con le predizioni;
- il numero delle predizioni da calcolare.

Tutto il codice Java relativo a quest'oggetto è definito qui, il rimanente è codice C. La libreria libNwsWrapper.so è ottenuta grazie ai seguenti sorgenti:

- NwsWrapper.c
- it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper.h
- config_portability.h, osutil.h

I file rilevanti sono i primi due, gli ultimi invece sono dei file di intestazione che contengono informazioni relative ai metodi di previsione delle librerie del Network Weather Service e sono stati recuperati direttamente dai sorgenti originali di NWS. Mostriamo il codice del file *it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper.h:*

```
#include <jni.h>                                                          1
                                                                          2
#ifndef _Included_it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper          3
#define _Included_it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper          4
#ifdef __cplusplus                                                        5
extern "C" {                                                              6
#endif                                                                    7
/*                                                                        8
 * Class:     it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper              9
 * Method:    addForecast                                                 10
 * Signature: ([Lit/unipmn/dcs/predictor/nws/NwsMeasurement;Lit/unipmn/dcs/ 11
  predictor/nws/NwsPredictions;I)V
 */                                                                       12
JNIEXPORT void JNICALL                                                    13
 Java_it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper_addForecast
  (JNIEnv *, jobject, jobjectArray, jobject, jint);                       14
                                                                          15
#ifdef __cplusplus                                                        16
}                                                                         17
#endif                                                                    18
#endif                                                                    19
```

Questo file d'intestazione viene generato automaticamente al momento della compilazione della classe Java *it.unipmn.dcs.predictor.nws.wrapper.NwsWrapper* e dev'essere incluso nel file C che implementa il metodo nativo, è possibile

notare come ogni oggetto Java sia rappresentato da un equivalente C con la seguente corrispondenza:

- jobject / it.unipmn.dcs.predictor.nws.wrapper.NwsWrapper
- jobjectArray / it.unipmn.dcs.predictor.nws.NwsMeasurement []
- jobject / it.unipmn.dcs.predictor.nws.NwsPredictions
- jint / Java "int" type

Si mostra ora il codice invece di NwsWrapper.c

```
#include "it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper.h"
#include "osutil.h"
#include <nws_api.h>

double jgetMeasurementElement (JNIEnv *env, jobjectArray
 joaMeasurementsArray, char *name, int i) {...}

unsigned jsetPredictions (JNIEnv *env, jobject joPredictions,
                          double timeStamp,
                          double measurement,
                          double forecast,
                          double error) {...}

JNIEXPORT void JNICALL
 Java_it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper_addForecast
  (JNIEnv *env, jobject jo, jobjectArray joaMeasurementsArray, jobject
   joPredictions, jint nLastPredictions) {...}
```

Si evidenzia l'inclusione del file header che permette al compilatore C di creare delle librerie che comunicano tramite la Java Virtual Machine per lo scambio dei vari oggetti e delle diverse variabili. Quest'insieme di sorgenti C compilati formano la libreria libNwsWrapper.so che viene poi chiamata da Java, ed ottenuta tramite i seguenti parametri del compilatore C:

```
gcc -g -Wall -I/usr/include -I/usr/lib/jvm/java-1.5.0-sun/include -I/usr/
 lib/jvm/java-1.5.0-sun/include/linux -shared -o libNwsWrapper.so NwsWrapper
 .c -lm -lnws
```

Questo comando specifica al compilatore tramite le inclusioni:

```
-I/usr/lib/jvm/java-1.5.0-sun/include -I/usr/lib/jvm/java-1.5.0-sun/include/
 linux
```

che la libreria condivisa risultante farà uso delle Java Native Interface (JNI). Il metodo *jgetMeasurementElement()* é servito a nascondere il metodi C/JNI e semplicemente ritorna la singola misurazione. Il metodo *jsetPredictions()* è servito analogamente a fare la stessa cosa per la conversione dai tipi C e la memorizzazione nell'oggetto java NwsPredictions. Il lavoro vero è proprio viene fatto dalla *Java_it_unipmn_dcs_predictor_nws_wrapper_NwsWrapper_addForecast* che effettua la previsione inserendo i dati relativi alle misurazioni e previsioni dentro le strutture dati definite da NWS:

- NWSAPI_Measurement
- NWSAPI_Forecast
- NWSAPI_ForecastCollection

definite queste è possibile ricavare le previsioni tramite il seguente codice:

```
for(c = 0; c < measurementsSize; c++) {
    measurement.timeStamp = jgetMeasurementElement (env,
     joaMeasurementsArray, "getTimeStamp", c);
    measurement.measurement = jgetMeasurementElement (env,
     joaMeasurementsArray, "getMeasurement", c);
    UpdateForecastState
        (forecastState, &measurement, 1, &forecast, 1);
    if (c >= measurementsSize - nLastPredictions)
        jsetPredictions (
            env,
            joPredictions,
            (double) forecast.measurement.timeStamp,
            (double) forecast.measurement.measurement,
            (double) forecast.forecasts[MSE_FORECAST].forecast,
            (double) forecast.forecasts[MSE_FORECAST].error);
}
```

Dato un numero definito di misurazioni, queste vengono inserite dentro una struttura dati *NWSAPI_Measurement* e si calcolano le relative predizioni inserendole dentro una struttura dati *NWSAPI_ForecastCollection* e richiamando il metodo *UpdateForecastState*. Concludo questa sezione spiegando che questa implementazione "ibrida" è stata scelta per evitare di riscrivere da zero i metodi di previsione di NWS, e per fornire una piena compatibilità con i suoi vari strumenti.

3.2.5 Predittore Lineare

Il predittore lineare implementato all'interno della nostra libreria fornisce previsioni tramite le tecniche di predizione lineare descritte in 2.1. L'intero predittore è scritto interamente in linguaggio Java e le sue classi sono contenute all'interno del package *it.unipmn.dcs.predictor.linear*.

Le classi che compongono il predittore sono le seguenti:

- LinearMeasurement
- LinearMeasurements
- LinearPrediction
- LinearPredictions

- LinearPredictor
- LinearPredictorMath

Come nel caso precedente la classi *LinearMeasurement* e *LinearMeasurements* implementano i metodi relativi alla memorizzazione dei valori numerici relativi alle misurazioni ed i metodi relativi all'ottenimento di questi da diversi input. Vediamo il codice relativo alla singola misurazione:

```
public class LinearMeasurement implements Measurement {                    1
                                                                            2
    private double timeStamp;                                               3
    private double measurement;                                             4
                                                                            5
    public LinearMeasurement(double timeStamp, double measurement) {...}   6
    public LinearMeasurement() {...};                                       7
    public double getTimeStamp() {...}                                      8
    public double getMeasurement() {...}                                    9
    public void setTimeStamp(double timeStamp) {...}                        10
    public void setMeasurement(double measurement) {...}                    11
                                                                            12
    public String toString() {...}                                          13
}                                                                           14
```

come si può notare la struttura è molto simile a quella della NwsMeasurement ed infatti a questa seconda ci siamo ispirati. La singola misurazione viene identificata dal *timestamp* di prelevamento e dal valore misurato *measurement*. Seguono i vari metodi per il recupero e l'inserimento di questi valori e per stamparli in un formato adeguato. Analogamente abbiamo la *LinearMeasurements* contenitrice dei singoli oggetti LinearMeasurement:

```
public class LinearMeasurements implements Measurements {                  1
                                                                            2
    private List <Measurement> list;                                        3
                                                                            4
    public LinearMeasurements(String filePath) throws IOException{...}      5
    public LinearMeasurements(List <Measurement> list) {...}                6
    public void getFromFile(String filePath) throws IOException,            7
     NumberFormatException {...}
    public List <Measurement> getList() {...}                               8
    public boolean isValid() {...}                                          9
}                                                                           10
```

La classe contiene i metodi per la memorizzazione di una lista di oggetti LinearMeasurement e per il loro ottenimento, inoltre contiene un metodo per controllare la validità di questa lista di misurazioni. Le classi *LinearPrediction* e *LinearPredictions* implementano i metodi relativi alla memorizzazione dei valori numerici predetti e delle loro informazioni accessorie. Vediamo l'oggetto LinearPrediction:

```
public class LinearPrediction implements Prediction {                      1
                                                                            2
    private double timeStamp;                                               3
    private double measurement;                                             4
    private double forecast;                                                5
```

```
    private double error;
    ......
}
```

anche in questo caso si hanno quattro tipi double che contengono il *timeStamp* ed il valore dell'ultima misurazione *measurement* relativo all'ultima misurazione, il valore predetto *forecast* e l'errore stimato *error*. La classe contenitore *LinearPredictions* è analoga a quella presentata per il predittore NWS e permette la memorizzazione di un numero N di predizioni, anche se nel nostro caso per ogni insieme di misurazioni viene prodotta una ed una sola predizione, questo al fine di ridurre il carico computazionale:

```
public class LinearPredictions implements Predictions {

    private List <Prediction> list;

    public LinearPredictions() {}

    public LinearPredictions(LinearPrediction prediction) {...}
    public LinearPredictions(double timeStamp, double measurement, double
     forecast, double error) {...}
    public LinearPredictions(String filePath) throws IOException{...}
    public void getFromFile(String filePath) throws IOException,
     NumberFormatException {...}
    public void addPrediction(LinearPrediction prediction) {...}
    public void addPrediction(double timeStamp, double measurement, double
     forecast, double error){...}
    public List <Prediction> getList() {...}
    public Prediction getLast() {...}
}
```

Al fine di realizzare un predittore efficiente si è cercato in rete ed in letteratura esempi di implementazioni relativi alla predizione lineare già esistenti. Dopo qualche ricerca si è giunti al *Numerical Recipes in C* [39], all'interno di questo testo era presente un codice scritto per il linguaggio C che implementava un metodo efficiente utilizzante la tecnica di Levinson e Durbin [17]. Si è proceduto quindi allo studio e alla comprensione del codice e a rimuovere le parti per noi non interessanti in quanto relative ad applicazioni sui segnali. Il risultato è stato quello di creare una classe che includesse tutti i metodi matematici relativi alla predizione chiamata *LinearPredictorMath* e riassumibile con il seguente codice:

```

public class LinearPredictorMath {

    private double xms;      // Mean Square Discrepancy.
    private double data[];       // A real vector of measured points.
    private double d[];          // Linear Prediction Coefficients.
    private int m;       // The number of Linear Prediction Coefficients to
     compute.
    private double future[];     // Future Data Points.
    private int nfut;        // The number of Future Data Points to compute.
    private boolean stability;  // Enable the stability of LP Coefficients.


```

```
    public LinearPredictorMath(double[] data, int m, int nfut, boolean
     stability) {
    }

    /**
     * Given a real vector of data[1..n], and given m, this routine returns
      m linear
     * prediction coefficients as d[1..m], and returns the mean square
      discrepancy as xms.
     */
    public void memcof() {...}

    /**
     * Given data[], an given the data's LP coefficients d[], this routine
     * predict the next nfut data points, which it returns in the array
     * future[1..nfut].
     */
    public void predic() {...}

    ....
}
```

Il costruttore della classe prende la serie di m misurazioni rilevate *data[]* ed un array di double dove vengono restituiti i "punti futuri" ossia le nostre previsioni. Si hanno poi diversi metodi che effettuano i vari passi computazionali per l'ottenimento delle previsioni, innanzitutto si ha il metodo *memcof* che date le misurazioni ritorna i coefficienti legati all'equazioni normali spiegate in 2.8, mentre il metodo *predic()* presi in input i coefficienti calcolati si occupa di stimare i punti futuri. É presente all'interno di questa classe una serie di metodi per l'inserimento e l'ottenimento dei diversi valori numerici utilizzati. La classe *LinearPredictor* delega quindi tutto il lavoro computazionale alla *LinearPredictorMath* e si limita a specificare ad essa la computazione di un singolo punto futuro, di una singola predizione. Dai nostri studi si é rilevato che il calcolo di predizioni successive alla prima oltre ad essere impreciso é fonte anche di lunghi ritardi computazionali ed é stato per questo motivo deciso di considerare solo la prima. Vediamo come questa classe richiama la LinearPredictorMath:

```
public class LinearPredictor implements Predictor{

    private LinearMeasurements measurements;
    private LinearPredictions predictions;

    public LinearPredictor (Measurements measurements, Predictions
     predictions) {...}
    public void setMeasurements(Measurements measurements) {...}
    public Predictions getPredictions() {...}

    public boolean compute(boolean stability) {
        int nfut = 1;   // number to future prevision

        if(measurements.isValid()) {

            double [] data = new double[measurements.getList().size()];
            double [] forecasts = new double[nfut];
```

```
            Iterator<Measurement> iterator = measurements.getList().iterator 17
             ();
            LinearMeasurement lastMeasurement = null;                          18
                                                                               19
            for(int i = 0; iterator.hasNext(); i++) {                          20
                lastMeasurement = (LinearMeasurement) iterator.next();         21
                data[i] = lastMeasurement.getMeasurement();                    22
            }                                                                  23
                                                                               24
            LinearPredictorMath lpm = new LinearPredictorMath(data, data.      25
             length - 1, nfut, stability);
                                                                               26
            lpm.predic();                                                      27
                                                                               28
            forecasts = lpm.getFuture();                                       29
                                                                               30
            predictions.addPrediction(                                         31
                    lastMeasurement.getTimeStamp(),                            32
                    lastMeasurement.getMeasurement(),                          33
                    forecasts[forecasts.length -1],                            34
                    lpm.getXms());                                             35
                                                                               36
            return true;                                                       37
        }                                                                      38
        else return false;                                                     39
                                                                               40
    }                                                                          41
    ....                                                                       42
}                                                                              43
```

Come precedentemente detto si richiede di calcolare tramite *nfut* una sola previsione. Queste due classi ed in particolare la precedente fanno uso di strutture dati per la rappresentazione di numeri complessi e risolutori di polinomi. Questi oggetti e metodi vengono forniti tramite la libreria *Apache Commons Math* che mette a disposizione vari metodi predefiniti per il calcolo numerico. L'implementazione delle tecniche di predizione lineare grazie a queste librerie e grazie agli algoritmi su cui ci siamo ispirati offre risultati equiparabili agli equivalenti scritti in codice C sia riguardo alla precisione di stima sia alla velocità computazionale.

3.2.6 Predittore Brevik

Il *BrevikPredictor* prende il suo nome dall'autore delle tecniche che utilizza per effettuare la previsione descritte in precedenza nella sezione 2.2. La realizzazione di questo predittore è stata possibile grazie al codice sorgente messo a disposizione dagli stessi autori, questo codice è scritto in linguaggio C e fa uso di diverse librerie native in quel linguaggio quali le:

- ANSI C Math
- GNU Multiprecision

per tali motivi abbiamo deciso di ricorrere nuovamente alle JNI. Questa soluzione si è mostrata vincente e la velocità computazionale e le prestazioni del predittore sono piuttosto alte. Anche in questo caso il predittore è caratterizzato da due package:

- it.unipmn.dcs.predictor.brevik
- it.unipmn.dcs.predictor.brevik.wrapper

che analogamente al caso di NWS contengono le classi base ed una classe "wrapper" che va a richiamare i metodi nativi in linguaggio C. Per il funzionamento di questo predittore è necessario che sulla macchina dove viene ospitato siano installate le due librerie precedenti. Iniziamo a descrivere gli oggetti del primo package, analogamente ai casi precedenti abbiamo le classi *BrevikMeasurement* e *BrevikMeasurements* utili al contenimento della singola misurazione raccolta e di una lista di queste. Mostriamo il codice della BrevikMeasurement:

```
public class BrevikMeasurement implements Measurement {                    1
                                                                            2
    private double timeStamp;                                               3
    private double measurement;                                             4
                                                                            5
    public BrevikMeasurement(double timeStamp, double measurement) {...}   6
    public BrevikMeasurement() {...};                                       7
    public double getTimeStamp() {...}                                      8
    public double getMeasurement() {...}                                    9
    public void setTimeStamp(double timeStamp) {...}                        10
    public void setMeasurement(double measurement) {...}                    11
                                                                            12
    public String toString() {...}                                          13
}                                                                           14
```

e la corrispondente BrevikMeasurements:

```
public class BrevikMeasurements implements Measurements {                  1
                                                                            2
    private List <Measurement> list;                                        3
                                                                            4
    public BrevikMeasurements(String filePath) throws IOException{...}      5
    public BrevikMeasurements(List <Measurement> list) {...}                6
    public void getFromFile(String filePath) throws IOException,            7
     NumberFormatException {...}
    public List <Measurement> getList() {...}                               8
    public boolean isValid() {...}                                          9
}                                                                           10
```

Come si può notare anche questo predittore nelle classi relative alle misurazioni presenta una struttura identica alle precedenti, questo all'inizio della progettazione non era previsto, in quanto ci aspettavamo che alcuni predittori potessero implementare tipi diversi di misurazioni e predizioni. Una futura modifica sarà creare una classe di misurazione standard comune a tutti e tre

le classi di misurazioni di questi predittori, in maniera tale da sfruttare l'ereditarietà Java ed avere un codice più snello e pulito. Il discorso appena fatto vale per le classi relative alle predizioni, ossia la *BrevikPrediction* e la *BrevikPredictions*, di cui elenchiamo il codice della prima:

```
public class BrevikPrediction implements Prediction {                    1
                                                                          2
    private double timeStamp;                                             3
    private double measurement;                                           4
    private double forecast;                                              5
    private double error;                                                 6
                                                                          7
    public BrevikPrediction(double timeStamp, double measurement, double 8
     forecast, double error) {...}
    public BrevikPrediction() {...}                                       9
    public double getTimeStamp() {...}                                    10
    public double getMeasurement() {...}                                  11
    public double getForecast() {...}                                     12
    public double getError() {...}                                        13
    public void setTimeStamp(double timeStamp) {...}                      14
    public void setMeasurement(double measurement) {...}                  15
    public void setForecast(double forecast) {...}                        16
    public void setError(double error) {...}                              17
    public String toString() {..}                                         18
}                                                                         19
```

e della seconda:

```
public class BrevikPredictions implements Predictions {                   1
                                                                          2
    private List <Prediction> list;                                       3
                                                                          4
    public BrevikPredictions(BrevikPrediction prediction) {...}           5
    public BrevikPredictions(double timeStamp, double measurement, double 6
     forecast, double error) {...}
    public BrevikPredictions(String filePath) throws IOException{...}     7
                                                                          8
    public void getFromFile(String filePath) throws IOException,          9
     NumberFormatException {...}
    public void addPrediction(BrevikPrediction prediction) {...}          10
                                                                          11
    public void addPrediction(double timeStamp, double measurement, double 12
     forecast, double error) {...}
    public List <Prediction> getList() {...}                              13
    public Prediction getLast() {...}                                     14
}                                                                         15
```

Riguardo alla classe BrevikPredictions nell'attuale implementazione conterrà soltanto un oggetto BrevikPrediction, in quanto abbiamo deciso di creare una sola previsione per volta, sempre per favorire una computazione più leggera. Andiamo ora a descrivere le classi principali che sono la *BrevikPredictor* e la *BreviKwrapper*, la prima è caratterizzata dal seguente codice:

```
public class BrevikPredictor implements Predictor {                      1
                                                                          2
    private BrevikMeasurements measurements;                             3
    private BrevikPredictions   predictions;                              4
                                                                          5
    public BrevikPredictor (Measurements measurements, Predictions       6
     predictions) {...}
```

```
    public void setMeasurements(Measurements measurements) {...}
    public Predictions getPredictions() {...}

    public boolean compute(double quantile, double conf) {
        if(measurements.isValid()){
            ...........

            BrevikWrapper brvkWrapper = new BrevikWrapper();
            double forecast = brvkWrapper.brevik_pred(measurementsArray,
             measurementsArray.length, quantile, conf);
            ...........
            predictions.addPrediction(
                    lastMeasurement.getTimeStamp(),
                    lastMeasurement.getMeasurement(),
                    forecast,
                    conf);

            return true;
        }
        else return false;
    }

    public boolean compute() {
        return this.compute(0.1, 0.95);
    }
}
```

come si può vedere il metodo *BrevikPredictor.compute()* ha la possibilità di inserire due parametri che sono un quantile *quantile* ed una confidenza *conf* relativi alla predizione calcolata. All'interno di questo metodo dopo aver recuperato gli oggetti relativi alle misurazioni e alle predizioni si effettua una chiamata alla classe BrevikWrapper:

```
public class BrevikWrapper {

    // Loading /usr/lib/libBrvkWrapper.so
    // TODO: Rember to insert the path of in a "Preference"
    static {
        try {
            System.loadLibrary("BrvkWrapper");
        } catch (UnsatisfiedLinkError aError) {
            System.out.println("Couldn't find the native library "
                    + aError.getMessage());
        } catch (SecurityException aError) {
            System.out.println("Wasn't allowed to load the native library "
                    + aError.getMessage());
        }
    }

    public native double brevik_pred(double [] list, int len, double quant,
     double conf);
}
```

che va a caricare inizialmente una libreria nativa in linguaggio C chiamata *libBrvkWrapper.so* che contiene un metodo nativo *brevik_pred()* a cui si passano:

- un array dei valore delle misurazioni *list*;

- la lunghezza di questa lista *len*;
- il quantile *quant*;
- la confidenza *conf*;

questi valori tramite le librerie JNI verranno convertiti in tipi C per essere utilizzati dalle funzioni della libreria libBrvkWrapper.so. Quest'ultima è ottenuta grazie a:

- BrevikWrapper.c
- it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper.h
- Sorgenti originali del predittore Brevik (in linguaggio C)

Si mostra ora il codice del file di intestazione di *it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper.h*:

```
#include <jni.h>                                                          1
/* Header for class it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper */ 2
                                                                          3
#ifndef _Included_it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper    4
#define _Included_it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper    5
#ifdef __cplusplus                                                        6
extern "C" {                                                              7
#endif                                                                    8
/*                                                                        9
 * Class:     it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper        10
 * Method:    brevik_pred                                                 11
 * Signature: ([DIDD)D                                                    12
 */                                                                       13
JNIEXPORT jdouble JNICALL                                                 14
 Java_it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper_brevik_1pred
  (JNIEnv *, jobject, jdoubleArray, jint, jdouble, jdouble);              15
                                                                          16
#ifdef __cplusplus                                                        17
}                                                                         18
#endif                                                                    19
#endif                                                                    20
```

questo file viene generato automaticamente al momento della compilazione della classe Java it.unipmn.dcs.predictor.brevik.wrapper.BrevikWrapper e dev'essere incluso nel file C che implementa il metodo nativo, si evidenzia l'equivalenza tra funzioni C e metodi Java:

- jobject / it.unipmn.dcs.predictor.brevik.wrapper.BrevikWrapper
- jdoubleArray / double [] list
- jint / int len
- jdouble / double quant

- jdouble / double conf

ed infine il codice di *BrevikWrapper.c*:

```
#include "it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper.h"
#include "brevik_pred.h"
#include <jni.h>

//double brevik_pred(double *list, int len, double quant, double conf);

JNIEXPORT jdouble JNICALL
 Java_it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper_brevik_1pred
  (JNIEnv *env, jobject jo, jdoubleArray jarray, jint len, jdouble quant,
   jdouble conf) {

    jboolean isCopy1;
    jdouble* list = (* env) -> GetDoubleArrayElements(env, jarray, &isCopy1)
     ;
    double result = brevik_pred(list, len, quant, conf);

    if (isCopy1 == JNI_TRUE) {
       (* env) -> ReleaseDoubleArrayElements(env, jarray, list, JNI_ABORT);
    }
    return result;
}
```

che si limita a chiamare le funzioni fornite da J. Brevik. L'insieme dei vari sorgenti C compilati formano la libreria libBrevikWrapper.so che viene poi chiamata da Java, questa viene ottenuta tramite la seguente configurazione del compilatore C:

```
# Makefile libBrvkWrapper.so

all : libBrvkWrapper.so

libBrvkWrapper.so : it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper.h
 norm.o mpf_comb.o jbindex.o jval.o jrb.o batch_q_sim.o rare_event.o
 brevik_pred.o
    gcc  -g -Wall -I/usr/include -I/usr/lib/jvm/java-1.5.0-sun/include -I/
     usr/lib/jvm/java-1.5.0-sun/include/linux -shared -o libBrvkWrapper.so
     norm.o mpf_comb.o jbindex.o jval.o jrb.o batch_q_sim.o rare_event.o
     brevik_pred.o BrevikWrapper.c -lm -lgmp

it_unipmn_dcs_predictor_brevik_wrapper_BrevikWrapper.h :
    javah -classpath ../bin it.unipmn.dcs.predictor.brevik.wrapper.
     BrevikWrapper

brevik_pred.o : brevik_pred.c brevik_pred.h
    gcc  -I./ -c brevik_pred.c

rare_event.o:   rare_event.c rare_event.h
        gcc  -I./ -c rare_event.c

batch_q_sim.o:  batch_q_sim.c batch_q_sim.h
        gcc  -I./ -c batch_q_sim.c

jrb.o:      jrb.c jrb.h
        gcc  -I./ -c jrb.c

jval.o:     jval.c jval.h
```

```
        gcc  -I./ -c jval.c                                    24
                                                               25
jbindex.o:  jbindex.c jbindex.h                                26
        gcc  -I./ -c jbindex.c                                 27
                                                               28
mpf_comb.o: mpf_comb.h mpf_comb.c                              29
        gcc   -I./ -c mpf_comb.c                               30
                                                               31
norm.o:     norm.c norm.h                                      32
        gcc  -I./ -c norm.c                                    33
```

Il rimanente codice descrive il funzionamento vero e proprio del predittore e non viene incluso qui in quanto non di creazione dell'autore di questa trattazione. Questa commistione di codice Java e C nonostante i primi dubbi si è comportata in maniera molto soddisfacente parimenti al caso di NWS e ha permesso di raggiungere in tempi brevi il nostro obiettivo principale che era quello di sperimentare i vari metodi di previsione messi a disposizione dalla letteratura accademica riguardante la disponibilità delle macchine in ambito Grid Computing.

3.2.7 Predittore Ibrido

L'ultimo predittore implementato è stato denominato *HybridPredictor* in quanto sfrutta un approccio ibrido, ispirandosi ad altri predittori di questo tipo che sfruttano più metodi di previsione per poi effettuarne un confronto e scegliere il risultato predetto che sembra in quel contesto migliore. Nel nostro caso la politica sfrutta i tre predittori sopra descritti

- NwsPredictor
- LinearPredictor
- BrevikPredittor

in tal maniera:

1. si prendono in considerazione l'insieme *data set* di tutte le misurazioni raccolte fino a quel momento di tempo e le si divide in due gruppi uno chiamato *training set* e l'altro *testing set*;
2. utilizzando l'insieme di training set si calcolano tre previsioni rispettivamente con i tre predittori sopra elencati;
3. viene scelta una predizione alla volta e si calcola il residuo rispetto ad ogni misurazione contenuta nell'insieme testingSet e si calcola l'*errore residuo medio*;

4. si seleziona come "vincente" la previsione che ha ottenuto il *minimo errore residuo medio*;

5. si calcola la previsione finale utilizzando il metodo di previsione vincente sopra tutto il data set.

Questo metodo è stato ispirato ad altri metodi ibridi discussi in letteratura e della sua validità discuteremo in seguito ora andiamo a vedere l'algoritmo sopra descritto come si traduce in codice Java. Le classi facenti parte del predittore ibrido sono contenute all'interno del package *it.dcs.unipmn.predictor.hybrid*. Sono presenti due classi per le misurazioni *HybridMeasurement* e *HybridMeasurements*, la prima non si discosta dalle varie implementazioni di Measurement:

```
public class HybridMeasurement implements Measurement {                    1
                                                                            2
    private double timeStamp;                                               3
    private double measurement;                                             4
                                                                            5
    public HybridMeasurement(double timeStamp, double measurement) {...}   6
    public HybridMeasurement() {...};                                       7
    public double getTimeStamp() {...}                                      8
    public double getMeasurement() {...}                                    9
    public void setTimeStamp(double timeStamp) {...}                        10
    public void setMeasurement(double measurement) {...}                    11
                                                                            12
    public String toString() {...}                                          13
}                                                                           14
```

mentre la seconda ha un aspetto come vedremo differente:

```
public class HybridMeasurements implements Measurements {                  1
                                                                            2
    private List <Measurement> list;                                        3
                                                                            4
    public HybridMeasurements(String filePath) throws IOException{...}      5
    public HybridMeasurements(List <Measurement> list) {...}               6
    public void getFromFile(String filePath) throws IOException,            7
     NumberFormatException {...}
    public List <Measurement> getList() {...}                               8
    public boolean isValid() {...}                                          9
                                                                            10
    public NwsMeasurements toNwsMeasurements() {...}                        11
    public LinearMeasurements toLinearMeasurements() {...}                  12
    public BrevikMeasurements toBrevikMeasurements() {...}                  13
}                                                                           14
```

Come è possibile vedere ci sono tre metodi non presenti nelle altri classi relative alle misurazioni ossia:

- toNwsMeasurements()
- toLinearMeasurements()
- toBrevikMeasurements()

che permettono di produrre le equivalenti classi per i predittori Nws, Linear e Brevik. Le classi di predizione *HybridPrediction* e *HybridPredictions* mantengono invece la stessa struttura utilizzata dagli altri predittori, vediamo la prima:

```
public class HybridPrediction implements Prediction {

    private double timeStamp;
    private double measurement;
    private double forecast;
    private double error;

    public HybridPrediction(double timeStamp, double measurement, double
     forecast, double error) {...}
    public HybridPrediction() {...}
    public double getTimeStamp() {...}
    public double getMeasurement() {...}
    public double getForecast() {...}
    public double getError() {...}
    public void setTimeStamp(double timeStamp) {...}
    public void setMeasurement(double measurement) {...}
    public void setForecast(double forecast) {...}
    public void setError(double error) {...}

    public String toString() {...}
}
```

e la seconda:

```
public class HybridPredictions implements Predictions {

    private List <Prediction> list;

    public HybridPredictions(HybridPrediction prediction) {...}
    public HybridPredictions(double timeStamp, double measurement, double
     forecast, double error) {...}
    public HybridPredictions(String filePath) throws IOException{....}
    public void getFromFile(String filePath) throws IOException,
     NumberFormatException {...}
    public void addPrediction(HybridPrediction prediction) {...}
    public void addPrediction(double timeStamp, double measurement, double
     forecast, double error) {...}
    public List <Prediction> getList() {...}
    public Prediction getLast() {...}
}
```

Si vede infine la classe che effettua la previsione, o meglio il confronto tra i vari metodi di previsione la *HybridPredictor*:

```
public class HybridPredictor implements Predictor {
    ...........
    public boolean computeHybrid() {
        ...........
        if (measurements.isValid()) {
            List<Measurement> dataSet = measurements.getList();
            if (dataSet.size() >= 2) {

                // Divido l'insieme di tutte le misurazioni in due parti
                trainingSet = dataSet.subList(0, dataSet.size() / 2);
                testingSet = dataSet.subList((dataSet.size() / 2) + 1,
                 dataSet
```

```
            .size());                                                    12
....                                                                     13
// Effettuo una previsione per tutte e tre                               14
method1 = new NwsPredictor(trainingSetMeasurements                       15
        .toNwsMeasurements(), nwsPredictions);                           16
method2 = new LinearPredictor(trainingSetMeasurements                    17
        .toLinearMeasurements(), linearPredictions);                     18
method3 = new BrevikPredictor(trainingSetMeasurements                    19
        .toBrevikMeasurements(), brevikPredictions);                     20
                                                                         21
if (method1.compute() && method2.compute() && method3.                   22
 compute()) {
    ....                                                                 23
    // Calcolo dei vari residui                                          24
    ....                                                                 25
    }                                                                    26
    linearRelativeE = linearRelativeE / testingSet.size();               27
    nwsRelativeE = nwsRelativeE / testingSet.size();                     28
    brevikRelativeE = brevikRelativeE / testingSet.size();               29
                                                                         30
                                                                         31
    if(nwsRelativeE < linearRelativeE && nwsRelativeE <                  32
     brevikRelativeE) {
        // Vince NwsPredictor e lo utilizzo per predirre.                33
        ....                                                             34
    }                                                                    35
    else if(linearRelativeE < brevikRelativeE) {                         36
        // Vince LinearPredictor e lo utilizzo per predirre.             37
        ....                                                             38
    }                                                                    39
    else {                                                               40
        // Vince BrevikPredictor e lo utilizzo per predirre.             41
        ....                                                             42
    }                                                                    43
    return true;                                                         44
} else                                                                   45
    System.err.print("Predictions computing error..");                  46
...........                                                              47
}                                                                        48
```

Si è evidenziato solo il metodo *HybridPredictor.computeHybrid()* che effettua il lavoro di confronto e determina il vincitore dei tre metodi utilizzandolo poi per computare la predizione finale. Questo predittore nonostante usi tre metodi di previsione differenti ha dei tempi buoni e nel capitolo successivo si evidenzia la sua accuratezza nelle predizioni.

3.3 Simulazione

Lo scopo principale del lavoro di tesi era realizzare un confronto tra diversi metodi di previsione ed uno studio di tale argomento al fine di scoprire quali fossero effettivamente i loro vantaggi e quali fossero le tecniche più efficienti. Per effettuare questo studio e questo confronto bisognava effettuare diverse simulazioni e sperimentazioni, in questa sezione mostriamo brevemente gli strumenti che sono stati realizzati al fine di produrre i dati che ci hanno

permesso di trarre le conclusioni sui vari metodi. Questi strumenti sono stati realizzati tramite brevi programmi scritti in linguaggio Java e tramite alcuni script Unix Shell e Python. L'idea era quella di creare velocemente degli strumenti utilizzabili altrettanto velocemente e non si è seguita una progettazione lineare ma aprendo delle sotto-progettazioni ogni volta che si affrontava un nuovo problema. Si sono creati principalmente tre insieme di strumenti:

- strumenti per la generazione delle misurazioni;
- strumenti per la generazione delle previsioni;
- strumenti per la generazione dei task;
- strumenti per il confronto statistico dei metodi di previsione.

Tutti questi strumenti si appoggiano su dei file di configurazione molto semplici e studiati per essere facilmente modificabili:

- simulation.conf;
- machines.conf.

Il file "simulation.conf" permette di specificare queste variabili:

```
export PRD_HOME=/home/guido/workspace/predictor
export PRD_SIM_HOME=$PRD_HOME/simulation
export PRD_LIB=$PRD_HOME/lib
export PRD_MEAS_FILES=$PRD_SIM_HOME/measurements
export PRD_MEAS_FILES_EXT=meas
export PRD_PRED_FILES=$PRD_SIM_HOME/predictions
export PRD_PRED_FILES_EXT=pred
export PRD_MACHINE_NUMBER=15
export PRD_METHOD=hybrid
export PRD_PREDICTIONS_CMD="$PRD_SIM_HOME/generate_predictions.sh"
export PRD_MEASUREMENTS_CMD="$PRD_SIM_HOME/generate_measurements.sh"
export PRD_MEASUREMENTS_DB=uniform
export PRD_MEASUREMENTS_SIZE=1001
export PRD_PREDICTIONS_SIZE=1001
export PRD_INITIAL_TIMESTAMP=0
export PRD_REPAIR_TIME=3600
export CLASSPATH=$PRD_HOME/bin:$PRD_HOME/lib/cernlite.jar:$PRD_HOME/lib/
 jssim-1.0.0-double.jar:$PRD_HOME/lib/commons-math-1.2-SNAPSHOT.jar
```

Questi parametri vengono specificati come variabili di ambiente UNIX, in maniera tale che poi possano essere modificati in tempo reale, nel caso servisse. I parametri relativi al tipo di distribuzione statistica da utilizzare per definire le simulazioni e così via vengono passati invece tramite riga di comando, all'interno di vari script, in maniera da avere un approccio più semplice e veloce. Si spiegano ora i parametri sopra descritti:

- PRD_HOME specifica la directory che contiene sia gli strumenti di simulazione che il codice Java del predittore, in maniera da poter operare modifiche velocemente ad entrambi in fase di sviluppo;
- PRD_SIM_HOME specifica la directory contenente tutti gli strumenti di simulazione;
- PRD_LIB specifica la directory contenente le librerie necessarie per la simulazione (nel nostro caso le librerie matematiche utilizzate);
- PRD_MEAS_FILES specifica la directory dove vogliamo generare i file di misurazione, oppure dove vogliamo copiare dei file di misurazione esistenti da utilizzare come input per la simulazione;
- PRD_MEAS_FILES_EXT specifica l'estensione dei file di misurazione, in maniera tale da poter utilizzare anche la stessa directory per misurazioni e predizioni;
- PRD_PRED_FILES specifica la directory dove verranno generati i file di previsione tramite il metodo di previsione prescelto;
- PRD_PRED_FILES_EXT specifica l'estensione dei file di previsione, in maniera tale da poter utilizzare anche la stessa directory per misurazioni e predizioni;
- PRD_MACHINE_NUMBER specifica il numero di macchine presenti all'interno della Grid da simulare;
- PRD_METHOD specifica il metodo di previsione da utilizzare potendo scegliere tra nws, linear, brevik ed hybrid;
- PRD_PREDICTIONS_CMD specifica il comando da eseguire per generare le predizioni all'interno della directory $PRD_PRED_FILES;
- PRD_MEASUREMENTS_CMD specifica il comando da eseguire per generare le misurazioni all'interno della directory $PRD_MEAS_FILES;
- PRD_MEASUREMENTS_DB specifica il tipo di distribuzione utilizzato per definire l'affidabilità simulata di una macchina e può assumere i valori uniform o weibull;
- PRD_MEASUREMENTS_SIZE specifica il numero di misurazioni raccolte o generate;

- PRD_PREDICTIONS_SIZE specifica il numero di predizioni da generare (nel caso di NWS si intende le misurazioni affiancate dalla singola previsione e dall'errore legato ad essa);
- PRD_INITIAL_TIMESTAMP specifica il timestamp di partenza da utilizzare per generare i file di misurazione;
- PRD_REPAIR_TIME specifica l'intervallo di tempo dopo il quale una macchina guasta ritorna operativa;
- CLASSPATH specifica il Java Classpath relativo ai package utilizzati per la simulazione.

Questo file è stato nel tempo costantemente aggiornato e modificato per poter specificare sempre un maggior numero di impostazioni personalizzate. Il secondo file di configurazione presente è *machines.conf* e presenta questa forma:

```
machine1 1 0.568664 477343                                    1
machine2 1.125 0.729215 856663                                2
machine3 1 0.570816 619497                                    3
machine4 1.4375 0.662438 610445                               4
machine5 1.125 0.560362 199690                                5
machine6 1 0.969769 1271536                                   6
machine7 1.125 0.729823 350024                                7
machine8 1.4375 0.677637 373307                               8
machine9 1 0.928094 753368                                    9
machine10 1.125 0.570816 619497                               10
...                                                           11
```

Questo file serve a definire le macchine all'interno della Grid, il primo valore identifica un nome fittizio per la macchina, il secondo valore è la potenza computazionale media (da 1 a 2 processori). Il secondo valore ed il terzo valore, sono gli shape e scale della distribuzione Weibull definiti sperimentalmente oppure i valori minimo e massimo di una distribuzione uniforme a seconda della configurazione selezionata. Definiti questi primi file di configurazione è possibile utilizzare gli script relativi alla generazione delle misurazioni e delle predizioni. É possibile inoltre utilizzare gli script per effettuare semplici analisi statistiche su di essi. Nel prossimo paragrafo vengono mostrati gli strumenti per la generazione delle misurazioni.

Strumenti per la generazione delle misurazioni

Al fine di generare le misurazioni sono stati creati due semplici script shell uno per la generazione delle misurazioni di una singola macchina, ed un altro per la generazione degli script di tutte le macchine:

- generate_measurements.sh

- genreate_meas_files.sh

Lo script *generate_measurements.sh* accetta dei parametri questi parametri sono:

- *startingTimestamp* indica il timestamp iniziale da affiancare alla misurazione;
- *measurementsSize* indica il numero di misurazioni che si vogliono generare tramite una distribuzione Weibull di parametri shape e scale;
- *repairTime* indica il tempo in secondi necessario che passa prima che una macchina diventata guasta ritorni disponibile;
- *alpha* indica il parametro shape della distribuzione Weibull o il valore minimo per una Uniforme;
- *beta* indica il parametro scale della distribuzione Weibull o il valor emassimo per una Uniforme;
- $[w/u]$ indica se le misurazioni devono essere generate tramite una distribuzione Weibull o Uniforme.

Ad esempio ipotizziamo di voler generare 10 misurazioni a partire dal timestamp 10000000 con un tempo di riparazione di un'ora, distribuite con una Weibull di parametri shape e scale pari a 0.568664 e 477343. Si digiterà quindi il seguente comando:

```
./generate_measurements.sh 1000000 10 3600 0.568664 477343
1000000 1552511330.137383
1003600 789958791.020958
1007200 4951411496.055393
1010800 46877294582.013410
...
```

Il secondo script per la generazione dei file ossia *generate_meas_files* non ha parametri. Dopo aver letto i file di configurazione simulation.conf e machines.conf genera all'interno della directory specificata da PRD_MEAS_FILES tutti i file di misurazioni per le singole macchine. Generati i file di misurazione si può ora creare i file di predizione da utilizzare per la simulazione, come mostrato nella successivo paragrafo.

Strumenti per la generazione delle predizioni

Analogamente a quanto fatto per la generazione delle misurazioni, esistono due script, uno per la generazione delle predizioni legati alla singola macchina, ed uno per la generazione delle predizioni di tutte le macchine:

- generate_prediction.sh
- generate_pred_files.sh

Il primo è un semplice script shell, che andando a leggere il metodo selezionato in *$PRD_METHOD* genera le previsione dati i file di misurazioni. Analogamente al caso delle misurazioni generate_pred_files.sh non ha ha parametri ed andando a leggere i file di configurazione simulation.conf e machines.conf genera all'interno della directory specificata da PRD_PRED_FILES tutti i file di predizione per le singole macchine. Generati i file di misurazione e previsione è possibile utilizzare degli script per l'analisi dell'efficacia del metodo di previsione.

Strumenti per la generazione delle statistiche

Si mostrano ora gli strumenti che permettono di analizzare quale sia l'efficacia del nostro predittore, di andare a guardare i dati statistici di ciascuna macchina quali: media delle misurazioni, valore vero di guasto per ciascuna macchina, media delle predizioni, valore predetto del prossimo guasto, errore assoluto ed errore relativo sulla previsione.

Generate N misurazioni m_i la *media delle misurazioni* è data da:

$$\frac{1}{N-1}\sum_{i=0}^{N-1} m_i \tag{3.3}$$

facendo l'assunzione che l'ultimo valore di una serie di misurazioni m_N sia il valore vero di guasto per ciascuna macchina. Date M predizioni p_i la *media delle predizioni* è data da:

$$\frac{1}{M}\sum_{i=0}^{M-1} p_i \tag{3.4}$$

Dato il valore vero di guasto per ciascuna macchina m_N e l'ultima previsione calcolata p_M si ha l'errore assoluto come

$$e_a = m_N - p_M \tag{3.5}$$

mentre si ha come errore relativo

$$e_r = e_a/m_N \tag{3.6}$$

Detto questo si ha lo script *compute_machine_stat.sh* che dato come parametro il nome della macchina va a cercare i file di misurazione e predizione nelle directory $PRD_MEAS_FILES e $PRD_PRED_FILES andando a generare i valori spiegati sopra, ad esempio:

```
[machine1]                                    1
Measurements Mean = 7261.829296               2
Real value = 9952.479483                      3
Prediction = 7261.829296                      4
Absolute error = 2690.650187                  5
Relative error 0.270350                       6
```

Al fine di poi verificare ed ordinare i dati in fogli di calcolo vi è uno script *compute_machine_stat_csv* che permette di generare delle tabelle sotto forma di *Comma Separated Value* (CSV), ad esempio:

```
./compute_machine_stat_csv.sh machine1                                    1
machine1.meas;7261.829296;9952.479483;7261.829296;2690.650187;0.270350    2
```

Sono stati creati anche due script per generare le statistiche relative a tutte le macchine e si chiamano rispettivamente *generate_all_statistics.sh* e *generate_all_statistics_csv.sh*. Qui si conclude la descrizione degli strumenti relativi all'analisi dell'accuratezza delle previsioni. Nel prossimo paragrafo si mostreranno gli strumenti per la generazione dei task da sottomettere alla Grid che andremo a simulare.

Strumenti per la generazione dei task

Si è deciso di creare degli strumenti che possano essere indipendenti tra loro, in maniera da poterli sfruttare con altri programmi o script. Seguendo questa idea è possibile generare delle informazioni relative a degli ipotetici task da sottomettere sulla Grid, queste informazioni vengono poi memorizzate in un file che verrà poi letto insieme ai file di configurazione da un simulatore ad eventi discreti per poi procedere alla simulazione. Sono stati creati due strumenti, uno permette la creazione di un unico file contenente una serie di task identificati dal *nome del task* e dal *tempo di esecuzione del task*, questo file è quello che servirà direttamente alla simulazione. L'altro strumento permette dato il file dei task appena descritto di generare un *job descriptor file* da passare ad Ourgrid nel caso si volesse realizzare una sperimentazione utilizzando questo strumento. Lo script per la generazione dei task si chiama *generate_tasks.sh* e permette di generare un numero arbitrario di task definito da una distribuzione scelta, avente un tempo di esecuzione arbitrario generato da un'altra distribuzione scelta. Lo script prende questi parametri:

- *tnDistributionType* specifica il tipo di distribuzione da usare per determinare il numero di task da sottomettere alla macchina, si può scegliere al momento tra Uniforme (1), Normale (2) e Weibull (3);
- *etDistributionType* specifica il tipo di distribuzione da usare per determinare i tempi di esecuzione del singolo task da sottomettere alla

macchina, si può scegliere al momento tra Uniforme (1), Normale (2) e Weibull (3);

- *task number parametes* è una serie di parametri per la generazione del numero di task dipendente dalla distribuzione scelta;
- *execution time parameters* è una serie di parametri per la generazione del tempo di esecuzione del task dipendente dalla distribuzione scelta;

I parametri relativi alle distribuzioni (task number parametes, execution time parameters) sono specificabili singolarmente ma sono gli stessi:

- nel caso si scelga una distribuzione uniforme sono i parametri di valore minimo e massimo;
- nel caso si scelga una distribuzione normale sono i parametri di media e deviazione standard;
- nel caso si scelga una distribuzione Weibull sono i parametri alpha e beta.

Ad esempio se si vuole generare un numero di task tramite una distribuzione uniforme di parametri $U(1, 10)$ e con dei tempi di esecuzioni determinati da una distribuzione Weibull di parametri $W(0.568664, 477343)$ si eseguirà lo script nella maniera seguente:

```
./generate_tasks.sh 1 2 1 10 0.568664 477343
task1 117575
task2 203463
task3 240251
task4 761702
task5 236859
task6 199700
task7 381412
task8 395294
task9 55103
```

Ipotizzando che si voglia poi in seguito sottomettere dei task tramite Ourgrid, lo script *generate_ourgrid_tasks.sh* permette di procedere nella seguente maniera:

```
./generate_tasks.sh 1 2 1 10 0.568664 477343 > task_file
./generate_ourgrid_tasks.sh task_file MyJobDescriptorFile > task_file.jdf
```

in tal maniera si genererà un file simile a questo:

```
job:
label: MyJobDescriptorFile
task:
remote: sleep task1 117575
task:
remote: sleep task2 203463
....
task:
remote: sleep task9 55103
```

Che potrà poi essere sottomesso da un client MyGrid per effettuare sperimentazioni e simulazioni tramite esso.

Simulatore ad eventi discreti

Durante lo studio delle tecniche di previsioni si è sentito il bisogno di vedere il comportamento del predittore all'interno di casi di allocazione dei task, si è quindi realizzato un semplice *simulatore ad eventi discreti*. Questo simulatore sfrutta le misurazioni, le previsioni ed i task generati precedentemente per generare i dati di simulazione. Al fine di velocizzare il procedimento di scrittura del simulatore si sono utilizzate delle apposite librerie Java chiamate *Simple Discrete-Event Simulation Library* (SSIM) che si sono andate a modificare leggermente per permettere l'utilizzo di tipi reali. Sull'implementazione ed il codice Java non si discute qui, si mostra semplicemente il funzionamento del simulatore a partire dallo script che lo esegue il *run_simulation.sh* che prende come parametri:

- *tasksListFile* il file contenente la lista dei task e dei loro tempi di esecuzione;
- *machinesConfFile* il file contenente i dati relativi alla potenza computazionale delle macchine e alla loro affidabilità;
- *faultsListFile* il file che contiene la lista dei guasti simulati per le macchine;
- *repairTime* il tempo di riparazione che necessita una macchina per tornare attiva.

Dati questi parametri il simulatore può partire e fornire dati, per ogni macchina simula i seguenti eventi:

- *MACHINE_UP* la macchina viene accesa e si notifica come inattiva;
- *MACHINE_IDLE* la macchina è inattiva e quindi può ricevere task da elaborare;
- *MACHINE_BUSY* la macchina ha ricevuto un task da eseguire e si rende occupata per gli altri;
- *MACHINE_DOWN* la macchina ha avuto un guasto o è stata spenta ed è quindi inutilizzabile;

mentre i task sono rappresentati da due eventi:

- *TASK_SUBMIT* un determinato task viene sottomesso ed allocato su una macchina inattiva a seconda della politica di allocazione;
- *TASK_FINISHED* un determinato task viene concluso da una determinata macchina e rimosso dalla simulazione;

se un task viene sottomesso su una macchina che cade prima della terminazione di questo, viene risottomesso. Normalmente il simulatore fornisce dei risultati sullo standard output piuttosto dettagliati e commentati come ad esempio:

```
[5609.65009] Machine(machine6) is down.                                    1
[5612.012238] Machine(machine9) is down.                                   2
[5831.824544] Machine(machine12), with Task(task3) not completed, is down. 3
[5835.0] Task(task3) submitted on Machine(machine5).                       4
[5835.0] Machine(machine5) is busy.                                        5
[5874.828673] Machine(machine8) is up.                                     6
[5874.828673] Machine(machine8) is idle.                                   7
[5905.86149] Machine(machine2), with Task(task4) not completed, is down.   8
[5910.0] Task(task4) submitted on Machine(machine8).                       9
[5910.0] Machine(machine8) is busy.                                        10
[5914.533096] Machine(machine3) is down.                                   11
[6035.583028] Machine(machine15) is up.                                    12
```

questi però non permettono una visione globale e complessiva, si è quindi aggiunta la possibilità di generare anche in questo caso un foglio di calcolo in formato CSV. Le funzionalità offerte dal simulatore sono in sintesi quelle appena presentate, esso risulta molto semplice ma ci ha permesso di ottenere comunque delle stime della validità di metodi di allocazione basati sulla previsione dell'affidabilità.

Capitolo 4

Analisi dei risultati

Quest'ultimo capitolo descrive i risultati dei nostri studi nel campo della previsione dell'affidabilità in ambito Desktop Grid e dell'applicabilità di questa all'interno delle politiche di scheduling. Questa fase è interamente sperimentale ed utilizza gli strumenti che sono stati studiati ed implementati nei capitoli precedenti. Si mostrerà qual'è l'accuratezza dei metodi statistici nel predirre il guasto di una macchina. Descritti questi si presenteranno invece i risultati di alcune simulazioni condotte per vedere quali vantaggi porta un algoritmo knowledge-aware che usi l'affidabilità. É importante sottolineare che anche il miglior metodo di previsione può essere utilizzato in maniera erronea e non garantire una buona schedulazione. Questo spiega come l'accuratezza di una informazione e come questa viene utilizzata all'interno di una politica debbano essere trattati come argomenti ortogonali. Si inizierà mostrando i parametri con cui si sono condotte le sperimentazioni, seguiti dai risultati ed infine dalle riflessioni suscitate dal confronto dei dati.

4.1 Scenari di sperimentazione

Per effettuare le nostre sperimentazioni si è tenuto conto di quattro scenari di sperimentazione, ossia quattro diverse configurazioni di Desktop Grid. Tali configurazioni sono state poi usate per simulare il comportamento dello sche-

	Bassa Affidabilità	*Alta Affidabilità*
Bassa Potenza Comp.	Scenario 2	Scenario 1
Alta Potenza Comp.	Scenario 4	Scenario 3

Tabella 4.1: Tipologie di Scenario

duler e dei vari metodi di previsione. Questi quattro scenari si differenziano per l'*affidabilità effettiva delle macchine* e per la *potenza computazionale* di queste. Combinando questi due valori si sono ottenuti i quattro scenari (vedi Tabella 4.1):

- lo *scenario1* è caratterizzato da una bassa potenza computazionale e da macchine molto affidabili;
- lo *scenario2* è caratterizzato da una bassa potenza computazionale e da macchine poco affidabili;
- lo *scenario3* è caratterizzato da un'alta potenza computazionale e da macchine molto affidabili;
- lo *scenario4* è caratterizzato da un'alta potenza computazionale e da macchine poco affidabili.

É importante ricordare che per noi ed in letteratura l'affidabilità di una macchina è indicata dal rapporto tra i tempi di uptime e quelli di downtime della macchina.

La Grid da noi simulata dispone di quindici macchine. Per quanto concerne i tempi di guasti adottati per gli scenari con macchine molto affidabili si sono generati i diversi valori tramite una distribuzione Weibull i cui parametri shape e scale sono stati presi in prestito dai valori raccolti all'interno dell laboratorio *Computer Science Instructional Laboratory* (CSIL) dell'Università della California presso Santa Barbara (vedi Tabella 4.2). Nei due scenari con macchine caratterizzate da una bassa affidabilità si sono calcolati i tempi di disponibilità tramite una distribuzione uniforme $U(MeanExeTime * 0.5, MeanExeTime * 1.5)$ dove $MeanExeTime$ è il tempo medio di esecuzione stimato dei task rispetto alla potenza computazionale media delle macchine.

In Tabella 4.3 si hanno i diversi scenari con le distribuzioni usate per generare i valori legati alla variabilità della potenza computazionale $CpuPower$, dei vari tempi di guasto delle macchine $DispoVary$ ed infine dei tempi di riparazione $RepairTime$. Queste informazioni sono usate sia per effettuare le simulazioni legate alla precisione dei metodi di previsione sia per quelle relative al loro utilizzo all'interno di politiche di schedulazione. Nella prossima sezione si mostra il confronto tra l'accuratezza dei diversi metodi di previsione.

Macchina	*CPU*	*shape*	*scale*
machine1	1	0.568664	477343
machine2	1.125	0.729215	856663
machine3	1	0.570816	619497
machine4	1.4375	0.662438	610445
machine5	1.125	0.560362	199690
machine6	1	0.969769	1271536
machine7	1.125	0.729823	350024
machine8	1.4375	0.677637	373307
machine9	1	0.928094	753368
machine10	1.125	0.570816	619497
machine11	1.4375	0.607016	405233
machine12	1	0.616905	228117
machine13	1.125	0.537631	178959
machine14	1.4375	0.68684	248058
machine15	1.4375	0.556867	243961

Tabella 4.2: Risorse CSIL

Scenario	*CpuPower*	*DispoVary*	*RepairTime(s)*
scenario1	$CpuPower = U(1, 2)$	$Weibull(shape, scale)$	300s
scenario2	$CpuPower = U(1, 2)$	$U(12000, 36000)$	3600s
scenario3	$CpuPower = N(5, 1.5)$	$Weibull(shape, scale)$	300s
scenario4	$CpuPower = N(5, 1.5)$	$U(3600, 10800)$	3600s

Tabella 4.3: Scenari di simulazione

4.2 Confronto dell'accuratezza dei metodi di previsione

In questa sezione descriviamo come sono state condotte le sperimentazioni per l'analisi dell'accuratezza dei diversi metodi di previsione e quali sono stati i risultati ottenuti. I vari metodi di previsione in letteratura fanno uso dell'*insieme delle misurazioni pregresse* relative ai tempi di uptime delle macchine per determinare una previsione sul prossimo guasto di un calcolatore. Si è quindi proceduto a generare diversi insiemi di misurazioni per ciascuna macchina di ciascuno scenario. In particolare, date le distribuzioni inerenti l'affidabilità definite per ciascuno scenario si sono generati diversi tempi di guasto, più precisamente:

Guasti generati	20	100	300	500	700	1000
Numero di misurazioni minime (20%)	4	20	60	100	140	200

questo significa che per un totale di guasti generati n, si è usato il $n*20\% = m$ di questi per generare la prima previsione p_m, che è stata confrontata con il tempo di guasto f_{m+1} ricavandone un errore assoluto:

$$ErrAss_m = p_m - f_{m+1} \tag{4.1}$$

ed un errore relativo:

$$ErrRel_m = ErrAss_m / f_{m+1} \tag{4.2}$$

questi valori si calcolano incrementalmente fino ad ottenere $n - m$ predizioni con i relativi errori assoluti e relativi. Questa metodologia è stata scelta per studiare l'andamento dell'accuratezza della previsione all'aumentare dell'insieme delle misurazioni. Ad esempio relativamente al metodo *NWS* e ad una determinata macchina per il caso con 20 misurazioni si avrà una tabella di questo tipo:

Guasti	Disponibilità	Predizione	ErrAss	ErrRel
4	35463066.9	7245794179.66	-7210331112.75	-203.32
5	27161660204.82	6371225004.06	20790435200.77	0.77
6	8780249359.17	6659425581.78	2120823777.38	0.24
...	...	...	...	...
19	40105928828.18	8107895125.28	31998033702.9	0.8
20	3598151669.73	9707796810.43	-6109645140.69	-1.7

Si è cerca ora di riassumere la grossa mole di dati ottenuta in poche tavole rappresentative. Inizialmente si è pensato di utilizzare la *media aritmetica*

dei diversi valori assoluti e relativi per ciascuna macchina ed infine la media di queste medie per ciascun caso di campionamento.

Purtroppo la media non mostra un risultato significativo in quanto molto spesso tra gli errori ci sono alcuni *outlier*, ossia alcuni valori marginali che si discostano enormemente dal comportamento abituale. Per risolvere questo problema si è deciso di usare la *mediana* al posto della media per la sua proprietà di fornire un buon indicatore del comportamento medio di un campione indipendentemente dagli outlier.

Si osservi innanzitutto per il primo scenario la Tabella 4.4. Il primo scena-

20	Nws	Linear	Brevik	Hybrid
ErrAss	808202499.02	574856150.81	521831017.17	808202499.02
ErrRel	0.99	1.28	**0.98**	0.99
100	Nws	Linear	Brevik	Hybrid
ErrAss	406581601	621703885.56	281702702.7	294782926.31
ErrRel	**0.98**	1.86	0.99	0.99
300	Nws	Linear	Brevik	Hybrid
ErrAss	526394654.5	937252710.37	306085941.52	313152173.68
ErrRel	1.28	2.15	0.99	**0.99**
500	Nws	Linear	Brevik	Hybrid
ErrAss	607490392.43	817293966.52	335201062.2	338361999.63
ErrRel	1.32	1.98	0.99	**0.99**
700	Nws	Linear	Brevik	Hybrid
ErrAss	630661531.62	955426047.16	326669786.35	337237755.04
ErrRel	1.33	2.33	0.99	**0.99**
1000	Nws	Linear	Brevik	Hybrid
ErrAss	534305620.01	876669838.43	327093354.53	328661286.48
ErrRel	1.48	2.43	0.99	**0.99**

Tabella 4.4: Accuratezza delle previsioni per lo Scenario 1

rio è caratterizzato da una bassa eterogeneità nella potenza computazionale delle varie macchine. Dai dati si può notare come in media le previsioni di tutti e quattro i metodi di previsione siano quasi sempre errate. Questa mancanza dei vari predittori era già sospettata all'inizio del nostro studio che è nato, tra le altre cose, al fine di verificare questa debolezza dei vari metodi presenti in letteratura. Dai dati si è quindi confermato il fatto che i metodi di previsione sui guasti sono pressochè inutili quando si hanno macchine molto affidabili, ed è molto difficile fornire una buona previsione. I due metodi che si sono comportati meglio sono le previsioni tramite il metodo

proposto da J. Brevik ed utilizzante l'MLE ed il nostro predittore ibrido. I due metodi che si sono comportati peggio sono risultati la predizione lineare e NWS che oltre ad avere degli errori maggiori al 99% hanno rilevato una tendenza a sbagliare ancora di più con l'aumentare del numero di misurazioni a disposizione. Questo risultato mostra che l'avere un insieme di misurazioni modesto o un insieme di misurazioni più importante risulta indifferente come accuratezza in questo tipo di metodi. Risultati decisamente migliori si sono

20	Nws	Linear	Brevik	Hybrid
Errore assoluto	458700245.8	620880076.72	451858602.97	358504933.53
Errore relativo	**0.91**	1.32	0.96	0.99
100	Nws	Linear	Brevik	Hybrid
Errore assoluto	499756987.92	1016734981.81	285474367.88	316649595.35
Errore relativo	**0.9**	1.79	0.99	0.99
300	Nws	Linear	Brevik	Hybrid
Errore assoluto	558753408.89	904149554.53	346749119.55	346749119.55
Errore relativo	1.16	2.08	0.99	**0.99**
500	Nws	Linear	Brevik	Hybrid
Errore assoluto	663455234.24	1165517147.36	345333949.14	364751695.99
Errore relativo	1.43	2.45	0.99	**0.99**
700	Nws	Linear	Brevik	Hybrid
Errore assoluto	624534634.13	917809123.51	337200030.28	337765089.17
Errore relativo	1.69	2.46	0.99	**0.99**
1000	Nws	Linear	Brevik	Hybrid
Errore assoluto	547913664.45	754048285.57	245071435.46	249137946.47
Errore relativo	1.51	2.33	0.99	**0.99**

Tabella 4.5: Accuratezza delle previsioni per lo Scenario 3

rilevati negli scenari 2 e 4 caratterizzati da macchine poco affidabili e con lunghi tempi di riparazione. La Tabella 4.6 mostra i risultati ottenuti per lo Scenario 2. In questo scenario l'affidabilità varia seguendo una distribuzione uniforme $U(12000, 36000)$ dove le macchine quindi sono accese da un minimo di 3 ore ad un massimo di 10 e dov'è richiesta un ulteriore ora prima che una macchina guasta ritorni operativa. L'errore relativo massimo rilevato è stato di 0.28 ed in questo scenario i metodi di previsione che si sono dimostrati più efficaci sono stati NWS ed il nostro metodo ibrido, seguiti con poco scarto dai risultati forniti con la previsione lineare. Inoltre si può osservare che l'incremento dell'insieme delle misurazioni pregresse corrisponde ad un incremento dell'accuratezza dei migliori predittori per questo scenario. Ad

20	Nws	Linear	Brevik	Hybrid
Errore assoluto	6728.14	6938.28	10119.49	7775.47
Errore relativo	**0.25**	0.32	0.44	0.28
100	Nws	Linear	Brevik	Hybrid
Errore assoluto	6341.17	7092.2	10594.69	6821.65
Errore relativo	**0.25**	0.29	0.44	0.28
300	Nws	Linear	Brevik	Hybrid
Errore assoluto	6023.48	6651.45	10438.16	6470.42
Errore relativo	**0.24**	0.28	0.44	0.25
500	Nws	Linear	Brevik	Hybrid
Errore assoluto	5789.78	6585.18	10281.06	6276.15
Errore relativo	**0.23**	0.27	0.43	0.26
700	Nws	Linear	Brevik	Hybrid
Errore assoluto	5968.03	6632.72	10286.47	6342.58
Errore relativo	**0.24**	0.28	0.43	0.26
1000	Nws	Linear	Brevik	Hybrid
Errore assoluto	5892.86	6617.63	10243.02	6287.41
Errore relativo	**0.23**	0.27	0.43	0.25

Tabella 4.6: Accuratezza delle previsioni per lo Scenario 2

esempio, guardando gli errori relativi del miglior predittore (ossia NWS) si nota che a 20 misurazioni si ha un errore relativo di 0.25 che descresce fino a 0.23 quando il numero di misurazioni è pari a 500. Il metodo che si comporta peggio in questo scenario è quello di J.Brevik che presenta un errore relativo pari a 0.44. Andando ad analizzare il quarto scenario si ha un lieve miglioramento rispetto al precedente, probabilmente dovuto al fatto che l'affidabilità è inferiore rispetto allo scenario precedente. Nello Scenario 4 la distribuzione degli eventi di guasto si mostra con una uniforme $U(3600, 10800)$. Questo equivale a macchine che rimangono accese tra 1 e 3 ore, ossia uno scenario tipico di piccole reti domestiche o scolastiche. I dati rilevati sono mostrati in Tabella 4.7. Nello Scenario 4 il metodo di previsione migliore risulta NWS

20	Nws	Linear	Brevik	Hybrid
Errore assoluto	1979.97	2018.61	2973.76	2279.48
Errore relativo	**0.26**	0.3	0.44	0.33
100	Nws	Linear	Brevik	Hybrid
Errore assoluto	1835.16	2021.84	3141.14	1999.09
Errore relativo	**0.24**	0.27	0.43	0.28
300	Nws	Linear	Brevik	Hybrid
Errore assoluto	1802.96	2011.88	3005.44	1934.69
Errore relativo	**0.24**	0.27	0.43	0.25
500	Nws	Linear	Brevik	Hybrid
Errore assoluto	1818.88	1908.83	3042.72	1883.53
Errore relativo	**0.24**	0.27	0.43	0.25
700	Nws	Linear	Brevik	Hybrid
Errore assoluto	1826.71	2004.46	3031.17	1934.63
Errore relativo	**0.24**	0.27	0.43	0.26
1000	Nws	Linear	Brevik	Hybrid
Errore assoluto	1750.43	1983.44	3019.45	1880.89
Errore relativo	**0.24**	0.27	0.44	0.26

Tabella 4.7: Accuratezza delle previsioni per lo Scenario 4

che con 20 misurazioni sbaglia il 25% delle volte mentre da 100 misurazioni in poi risulta assestarsi su un errore del 24%. Buoni risultati vengono ottenuti con il predittore ibrido e con quello lineare. Anche in questo caso a classificarsi il metodo peggiore risulta il predittore Brevik, che si comporta meglio in scenari dove le macchine sono più affidabili. L'incremento delle dimensioni dell'insieme di misurazioni è determinante solo per li predittore lineare e quello ibrido, mentre come abbiamo già detto risulta per NWS ininfluente.

Alla fine dei nostri studi si è mostrato come non esista in letteratura un metodo a noi conosciuto che si comporti in maniera accettabile quando si tratta di rilevare guasti su macchine molto affidabili, mentre più o meno tutti i metodi si dimostrano efficaci in ambienti in cui i computer hanno un'affidabilità giornaliera, ossia rimangono accesi per un periodo non maggiore alle dodici ore. Ambienti di questo tipo sono per fortuna abbastanza diffusi in ambito Desktop Grid come ad esempio in una università dove i computer degli studenti e degli uffici rimangono accesi durante le ore di frequentazione dei docenti e del personale universitario. Purtroppo sono molto utili anche computer quali server o workstation o cluster in ambito Desktop Grid, ma per questi sembra non esserci al momento un buon modo per stimare correttamente l'affidabilità delle macchine.

Un risultato che ci ha soddisfatto è che il nostro metodo di previsione Ibrido non solo non risulta mai il peggiore ma anzi si classifica sempre vicino al metodo di previsione migliore per quel caso, discostandosene quasi sempre solo per la seconda cifra decimale dell'errore relativo. Questo conferma l'idea nata nell'implementarlo che consisteva nel cercare di fornire una previsione non ottima ma che modellasse il più possibile il comportamento passato dell'affidabilità della macchina.

4.3 Applicazione dei metodi di previsione nella schedulazione

L'utilità di avere un buon metodo di previsione dei guasti nella schedulazione in ambito Desktop Grid è quello di migliorare le prestazioni ed ottimizzare l'allocazione dei task. Ricordiamo che all'utilizzo di un metodo di previsione accurato deve corrispondere un'attenta scelta di utilizzo di questa informazione all'interno della politica di scheduling. Il riuscire a sottomettere dei task in macchine che conosciamo a priori essere affidabili è importante in quanto evita la risottomissione e quindi l'aumento sia dei tempi di completamento della BoT sia del traffico sulla rete. Tenendo conto di questi presupposti si sono effettuate delle simulazioni utilizzanti una politica di scheduling molto semplice che sfrutta solo le informazioni relative ai guasti delle macchine. La politica creata consta delle seguenti tre fasi:

1. si riordinano i task per tempo di esecuzione decrescente, dal più lungo al più corto;

2. si riordinano le macchine da quella stimata più affidabile a quella stimata meno affidabile;

3. si assegna la BoT ordinata al punto 1 alle macchine disponibili ordinate al punto 2.

Questa politica è stata poi applicata nei quattro scenari disponibili ognuno caratterizzato dalla presenza di quindici macchine, il tempo di ciascun task è stato generato con una distribuzione uniforme $U(18000, 54000)$ caratterizzata quindi da un tempo medio di esecuzione di 36000 secondi per una macchina di potenza computazionale 1. Per ogni scenario si sono sottomessi diversi gruppi di task variando quello che viene chiamato *Resource Rate (RR)*, definito da:

$$RR = \frac{NumeroDeiTask}{NumeroMacchineGrid} \tag{4.3}$$

Le dimensioni delle varie BoT calcolate tramite i parametri RR sono mostrate in Tabella 4.8. La simulazione prevede quindi la sottomissione di BoT di

Resource Rate	0.3	0.5	0.7	1	3	5	7	10	20	50
Task	5	8	11	15	45	75	105	150	300	750

Tabella 4.8: Resource Rate (RR)

dimensioni diverse, una per volta, sulle varie macchine. Ogni volta che una macchina esaurisce il suo tempo di uptime generato dalle varie distribuzioni, rimane guasta per un determinato tempo *RepairTime* e poi torna disponibile con un nuovo tempo di guasto. Le macchine continuano a spegnersi ed accendersi fino alla terminazione della BoT. Si memorizzano i tempi di completamento della stessa BoT a seconda che si sia usata la nostra politica tramite uno dei quattro metodi di previsione e tramite una politica di tipo Workqueue. Si effettua poi la differenza tra il tempo di completamento della BoT utilizzante una delle quattro politiche con quello fornito da una politica Workqueuee. Si ottengono così quattro valori per ciascun metodo. Se il valore è negativo allora la politica Workqueue ha impiegato meno tempo e l'aggiunta dell'informazione sull'affidabilità non è stata correttamente sfruttata, altrimenti l'aggiunta delle previsioni ha portato a prestazioni maggiori. La sezione si conclude discutendo i vantaggi dei vari metodi di schedulazione.

Si analizzano ora gli scenari caratterizzati da un'alta affidabilità delle macchine. Si parte dallo Scenario 1 in cui le macchine hanno una bassa eterogeneità computazionale ed i cui risultati della simulazione sono mostrati in Tabella 4.9. Il tempo di completamento massimo raggiunto in questo scenario, dove le macchine sono molto affidabili ma computazionale poco dotate e differenziate, è stato di 420 secondi. Si può notare come rispetto ad una politica Workqueue, dove i task vengono assegnati in maniera casuale alle varie

RR	Task	Nws	Linear	Brevik	Hybrid
0.3	5	-1.63	-1.65	-1.63	-1.63
0.5	8	0	-0.77	0	0
0.7	11	-1.5	-1.57	-1.5	-1.5
1	15	-3.03	-3.41	-3.03	-3.03
3	45	3.02	3.02	3.02	3.02
5	75	-1.78	-1.4	-1.74	-1.74
7	105	-1.74	-1.65	-1.76	-1.76
10	150	-3.16	-3.06	-3.11	-3.11
20	300	-5.01	-4.91	-5.04	-5.04
50	750	2.41	2.41	2.41	2.41

Tabella 4.9: Tempi di completamento relativi ad una Workqueue nello Scenario 1

macchine, ci sia un risultato di pochi secondi più inefficiente. Questo dipende più dalla configurazione dello scenario che dalla già non buona accuratezza dei vari metodi di previsione mostrata nella precedente sezione. Infatti essendoci poca varietà computazionale e poca potenza computazionale i task non vengono quasi mai risottomessi e prevale quindi la potenza delle macchine come variabile per il tempo di completamento dello scheduling. Una maggiore potenza e variabilità computazionale su macchine molto affidabili aumenta, anche se di poco, il vantaggio dato da una politica knowledge-aware basata sull'affidabilità come si riscontra in Tabella 4.10. Si studia ora l'allocazione

RR	Task	Nws	Linear	Brevik	Hybrid
RR	Lavori	Nws Ass.	Linear Ass.	Brevik Ass.	Hybrid Ass.
0.3	5	-1.6	0	0.21	0.21
0.5	8	1.18	2.35	2.35	2.35
0.7	11	-1.52	-0.88	-0.88	-0.88
1	15	-1.58	-0.24	-0.24	-0.24
3	45	0.02	0.11	0.08	0.08
5	75	0.73	0.63	0.62	0.62
7	105	0.62	0.58	0.71	0.71
10	150	0.43	0.48	0.41	0.41
20	300	0.59	1.71	0.57	0.57
50	750	0.97	0.97	0.97	0.97

Tabella 4.10: Tempi di completamento relativi ad una Workqueue nello Scenario 3

nei due scenari caratterizzati da una bassa affidabilità ossia lo Scenario 2 e 4. Lo Scenario 2 oltre ad essere caratterizzato da 15 macchine poco affidabili è caratterizzato anche da macchine computazionalmente molto simili. In tale scenario si è scelto di deviare l'affidabilità di una singola macchina scelta a caso. Si è modificata la sua distribuzione in maniera tale che fornisca valori molto affidabili generandola tramite un'uniforme $U(54000, 55000)$. Questa decisione è stata presa in quanto la generazione pseudo-casuale dei tempi di affidabilità delle macchine, spesso creava una Desktop Grid in cui alcuni task erano impossibili da completare, in quanto la loro esecuzione $\frac{execTime}{cpuPower}$ rimane comunque molto maggiore rispetto all'affidabilità stessa della macchina. In tale maniera vi è sempre una macchina molto affidabile che può completare anche i task più lunghi e mostrare effettivamente i guadagni portati da un metodo di previsione simile. In tale scenario i metodi di previsione vengono sfruttati al massimo ed infatti si ottengono dei risultati e dei risparmi fino a 10 minuti, come mostrato in Tabella 4.11. Nell'ultimo caso rappresentato

RR	Task	Nws	Linear	Brevik	Hybrid
RR	Lavori	Nws Ass.	Linear Ass.	Brevik Ass.	Hybrid Ass.
0.3	5	337.39	337.39	338.53	337.39
0.5	8	31.4	33.22	20.25	35.24
0.7	11	260.26	256.92	261.17	246.24
1	15	37.17	52.98	55.63	48.41
3	45	81.26	111.35	109.03	113.73
5	75	351.89	420.96	415.76	385.99
7	105	97.38	102.48	95.56	146.84
10	150	323.65	306.51	305.22	307.94
20	300	145.86	80.07	48.21	30.67
50	750	255.84	288.77	240.7	289.49

Tabella 4.11: Tempi di completamento relativi ad una Workqueue nello Scenario 2

dallo Scenario 4 i dati ottenuti confermano i ragionamenti precedentemente esposti. Questo scenario presenta macchine poco affidabili ma molto differenti dal punto di vista computazionale. Si riscontrano infatti delle prestazioni basse o inferiori alla poltica Workqueue in quei casi dove il numero dei lavori è piuttosto esiguo e dove quindi prevale la potenza computazionale sull'affidabilità. Aumentando invece il numero di task da completare aumentano seppur di poco le prestazioni portate dall'utilizzo di una politica knowledge-aware. I dati relativi allo Scenario 4 sono mostrati in Tabella 4.12. L'ultima

RR	Task	Nws	Linear	Brevik	Hybrid
0.3	5	0	0	0	0
0.5	8	-0.47	-0.47	-0.47	-0.47
0.7	11	4.02	4.02	4.02	4.02
1	15	-1.14	-1.14	-1.14	-1.14
3	45	-0.25	-0.83	-4.92	0.51
5	75	-0.14	-1.35	-0.71	0.08
7	105	-2.26	-2.69	-2.85	-2.34
10	150	3.38	4.75	3.38	4.75
20	300	3.26	3.26	3.26	3.26
50	750	1.27	1.27	-1.17	1.27

Tabella 4.12: Tempi di completamento relativi ad una Workqueue nello Scenario 4

riflessione da fare in questa sezione è relativa alle differenti prestazioni fornite dai vari metodi di previsione all'interno delle politiche di scheduling.

4.4 Riflessioni sull'uso di una politica informata sull'affidabilità

Si può notare osservando le quattro tabelle che gli scarti tra un metodo e l'altro sono di pochi secondi. Il metodo con le prestazioni generalmente inferiori risulta essere quello utilizzante la previsione lineare. Questo metodo è quello che fornisce tempi di completamento di diversi secondi superiori rispetto agli altri metodi. Lo scarto tra l'accuratezza dei due metodi di previsione etichettati come NWS e Brevik risulta piuttosto basso negli scenari interessanti (ossia quelli a bassa affidabilità), confermando le supposizioni fatte nella sezione precedente. Positivo è stato il risultato del predittore ibrido, che nella maggiorparte dei casi si classifica come il migliore, o come il secondo migliore. La spiegazione della "non-ottimalità" di questo metodo è semplice, il predittore ibrido mira a scegliere il metodo che si è adattato meglio all'interno della storia delle misurazioni di ciascuna macchina. Il metodo scelto non garantisce quindi di essere quello migliore per ogni previsione, ma garantisce di fornire una previsione che sarà giustificatamente vicina al comportamento mostrato dalla storia passata. I metodi di previsione disponibili attualmente in letteratura non permettono di effettuare buone predizioni a lungo tempo, si comportano invece in maniera discreta nel caso di previsioni a breve termine. Il risultato è quello di portare un incremento significativo

in quei casi dove appunto non vi sono macchine "affidabili". L'importanza di questo incremento è stata studiata solo tramite simulazioni, il futuro lavoro da fare sarà quello di integrare questi metodi all'interno di un middleware reale, come abbiamo già predisposto per OurGrid. Un ulteriore studio sarà relativo a come combinare in maniera opportuna il vantaggio portato dalla conoscenza e dalla previsione dell'affidabilità ad altri valori quali la potenza computazionale ed il traffico presente sulla rete, creando nuove politiche o integrando quelle già esistenti in letteratura.

Capitolo 5

Conclusione

5.1 Sommario dei contributi

La recente piattaforma di calcolo Desktop Grid sta riscuotendo sempre di più interesse. La possibilità di sfruttare al meglio la potenza di calcolo che altrimenti rimarrebbe inutilizzata è fonte di grande interesse in campo scientifico ed economico. Questo perché sempre più applicazioni necessitano di elevate potenze di calcolo difficilmente soddisfacibili dai singoli calcolatori. La capacità di sfruttare i tempi di inattività di reti già esistenti all'interno di una determinata organizzazione con semplicità e velocità si traduce in un risparmio economico, temporale e di risorse. A tutti questi vantaggi devono però corrispondere soluzioni intelligenti che si adattino ad un sistema distribuito che è caratterizzato da un'enorme eterogeneità. Questa eterogeneità riguarda sia la potenza computazionale che l'affidabilità delle macchine. L'obiettivo di questo lavoro di tesi ha riguardato principalmente il secondo aspetto attraverso lo studio di algoritmi di previsione dell'affidabilità delle macchine. I contributi riportati in questo ambito sono principalmente quattro:

Studio comparativo dei vari metodi di previsione. Un contributo di questa tesi è quello di fornire un compendio dei più interessanti metodi di previsione presentati nella letteratura scientifica fornendo uno strumento introduttivo per chi si avvicina a queste problematiche. L'aver sperimentato e discusso le caratteristiche dei vari metodi fornisce un indicazione su dove ci sia maggior bisogno di approfondire lo studio. Si è evidenziata e sperimentata la disparità di accuratezza tra metodi di previsione a "breve termine" e a "lungo termine", infatti si sono ottenuti buoni risultati nell'utilizzo dei primi e pessimi in quello dei secondi. Nel condurre questo studio si sono trattati inoltre i requisiti richiesti dal punto di vista delle misurazioni pregresse ne-

cessarie. Più precisamente si è effettuata un analisi su come la dimensione della storia passata pesi sull'esito di una predizione accurata.

Analisi dei vantaggi sull'uso di informazioni inerenti l'affidabilità delle macchine in politiche di scheduling knowledge-aware. Il possesso di uno strumento accurato risulta inutile se utilizzato in maniera erronea. É importante capire come utilizzare correttamente le informazioni relative alle macchine per ottenere un reale vantaggio all'interno delle politiche di scheduling. Si è compiuta un'analisi preliminare per "misurare" l'importanza di avere informazioni sull'affidabilità di una macchina all'interno di una politica di scheduling. Si sono proposti due metodi che mostrano come questa dimensione possa portare un interessante miglioramento delle prestazioni in termini di tempo di completamento medio dei task rispetto a politiche non informate. Nell'analisi e nella sperimentazione di tali politiche si è evidenziato il bisogno di affiancare l'utilizzo dell'affidabilità alle altre informazioni studiate. Come ad esempio suggerendo l'integrazione di politiche che trattino contemporaneamente l'affidabilità delle macchine e la potenza computazionale.

Crazione di un metodo di previsione ibrida. In questo lavoro, si è proposto un metodo nuovo per scegliere e discriminare tra i vari metodi di previsione in real time. Il predittore ibrido fornisce delle prestazioni interessanti e si differenzia dai suoi simili per intenti ed obiettivi. La predizione fornita cerca di adattarsi, imitare il comportamento passato e di non dare una soluzione ottima localmente. Tale metodo si ispira a tecniche e metodologie ben conosciute in altri ambiti quali la statistica ed il machine learning. L'implementazione proposta fornisce inoltre la possibilità di modificare il metodo in maniera pressoché immediata al fine di produrre estensioni future.

Creazione di una libreria per la predizione dell'affidabilità. La libreria da noi creata fornisce uno strumento riutilizzabile sia all'interno di sperimentazioni che riguardino l'affidabilità delle macchine, sia all'interno di politiche di scheduling per middleware scritte in Java. La sua ingegnerizzazione permette inoltre l'estensione di tale libreria e la sua integrazione in sistemi già esistenti. L'utilizzo di librerie native e l'integrazione tra codice C e Java fornisce inoltre buone prestazioni in termini di tempo di ottenimento della predizione.

5.2 Lavori futuri

Il lavoro di questa tesi ha lasciato in sospeso alcune problematiche interessanti descritte qui di seguito:

Creazione di metodi di previsione a lungo termine. L'aver studiato gli attuali metodi e la loro incapacità di prevedere i guasti di una macchina a lungo termine fornisce lo stimolo per la creazione e la risoluzione di queste problematiche. La ricerca e l'applicabilità di metodi statistici presi da altri paradigmi o ambiti scientifici potrebbe risolvere questo problema a cui la comunità scientifica sembra non aver ancora fornito risposte adeguate.

Studio ed implementazione di politiche knowledge-aware miste. Si è mostrato come l'utilizzo dell'affidabilità porti delle prestazioni nel tempo di completamento medio dei task all'interno di politiche di scheduling. É quindi necessario studiare un compromesso tra le varie informazioni disponibili ed utilizzabili dalle politiche di scheduling. In maniera tale da considerare contemporaneamente affidabilità e potenza di calcolo.

Nuovi metodi di previsione ibrida. Risulta interessante uno studio più approfondito in grado di combinare più metodi al fine di ricercare un legame tra meccanismi di predizione e caratteristiche delle macchine. La tendenza in letteratura è di creare metodi ibridi che riescano ad adattarsi e a scegliere autonomamente la tecnica giusta di previsione per quella determinata macchina. Questo si configura quindi come un campo aperto ed interessante per lo studio di nuove soluzioni.

Estensione e miglioramento della libreria di previsione. Si è interessati a migliorare a livello ingegneristico la libreria creata e ad estenderla opportunamente con nuovi metodi e nuove tecniche. Il primo passo sarà quello di rendere tale libreria di facile utilizzo per ulteriori studi.

Appendice A

Formule del Capitolo 2

A.1 Predizione Lineare

A.1.1 Matrice di Autocorrelazione

Il minimo errore quadratico totale, denotato da E_p, è ottenuto espandendo la (2.6) e sostituendoci la (2.8). Il risultato è il seguente

$$E_p = \sum_n s_n^2 + \sum_{k=1}^{p} a_k \sum_n s_n s_{n-k} \tag{A.1}$$

Si deve ora specificare l'intervallo della sommatoria su n in (2.6), (2.7) e (2.8). Per far questo si assume che l'errore in (2.6) è minimizzato sopra la durata $-\infty < n < \infty$, a questo punto si può ridurre la (2.8) e la (A.1) a

$$\sum_{k=1}^{p} a_k R(i-k) = -R(i), \qquad 1 \leq i \leq p \tag{A.2}$$

$$E_p = R(0) + \sum_{k=1}^{p} a_k R(k) \tag{A.3}$$

dove:

$$R(i) = \sum_{n=-\infty}^{\infty} s_n s_{n+1} \tag{A.4}$$

è detta *funzione di autocorrelazione* del segnale s_n. I coefficienti $R(i-k)$ formano quella che analogamente viene chiamata una *matrice di autocorrelazione*, che è una matrice simmetrica di Toeplitz, ossia una matrice dove tutti gli elementi lungo la diagonale sono uguali.

Si studia solitamente il segnale s_n sopra un intervallo finito, per fare questo un metodo è moltiplicare il segnale s_n per una funzione "finestra" w_n

al fine di ottenere un altro segnale $s_n^{'}$ che è zero fuori da un determinato intervallo $0 \leq n \leq N-1$:

$$s_n^{'} = \begin{cases} s_n w_n, & 0 \leq n \leq N-1 \\ 0, & altrimenti. \end{cases} \tag{A.5}$$

La funzione di autocorrelazione risulta ora data da

$$R(i) = \sum_{n=0}^{N-1-i} s_n^{'} s_{n+1}^{'}, \qquad i \geq 0 \tag{A.6}$$

A.1.2 Procedura ricorsiva di Durbin

Si espandi innanzitutto l'equazione A.2 nella seguente forma matriciale

$$\begin{bmatrix} R_0 & R_1 & R_2 & \dots & R_{p-1} \\ R_1 & R_0 & R_1 & \dots & R_{p-2} \\ R_2 & R_1 & R_0 & \dots & R_{p-3} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{p-1} & R_{p-2} & R_{p-3} & \dots & R_0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_p \end{bmatrix} = - \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ \vdots \\ R_p \end{bmatrix} \tag{A.7}$$

Si noti quindi che la matrice di autocorrelazione è una matrice di Toeplitz, ossia $p \times p$ è simmetrica e gli elementi lungo la diagonale sono identici. Osservato questo esiste una serie di metodi molto utilizzanti i più famosi sono quelli di Levinson [32], Robinson [40] e quello di Durbin [17]. Entrambi i metodi studiano il vettore colonna presente nella parte destra della (A.7). Tale vettore colonna include infatti tutti gli stessi elementi presenti nella matrice di autocorrelazione. Il metodo inoltre richiede solo $2p$ locazioni di memorizzazione e $p^2 + O(p)$ operazioni. La procedura ricorsiva di Durbin è la seguente:

$$E_0 = R(0) \tag{A.8a}$$

$$k_i = - \left[R(i) + \sum_{j=1}^{i-1} a_j^{i-1} R(i-j) \right] \Big/ E_{i-1} \tag{A.8b}$$

$$a_i^{(i)} = k_i \tag{A.8c}$$

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{i-1}, \qquad i \leq j \leq i-1 \tag{A.8d}$$

$$E_i = (1 - k_i^2) E_{i-1}. \tag{A.8e}$$

Le equazioni (A.8b)-(A.8e) vengono risolte ricorsivamente per $i = 1, 2, \dots, p$. La soluzione finale viene data da:

$$a_j = a_j^{(p)}, \qquad 1 \leq j \leq p.. \tag{A.8f}$$

A.1.3 Normalizzazione dei coefficienti

La soluzione di (A.7) non cambia se i coefficienti di autocorrelazione vengono scalati di una costante. In particolare, se tutti gli $R(i)$ sono normalizzati dividendo per $R(0)$, si hanno quelli che vengono conosciuti come i *coefficienti di autocorrelazione normalizzati* $r(i)$:

$$r(i) = \frac{R(i)}{R(0)} \tag{A.9}$$

per cui vale la proprietà $|r(i)| \leq 1$.

Un sotto prodotto del calcolo della soluzione in (A.8) è il calcolo dell'errore minimo totale E_i ad ogni passo. Si può mostrare facilmente che l'errore minimo E_i decresce (o rimane lo stesso) parimenti alla crescita dell'ordine del predittore. E_i non è mai negativo in quanto elevato al quadrato. Si ha quindi

$$0 \leq E - i \leq Ei - 1, \qquad E_0 = R(0). \tag{A.10}$$

Se i coefficienti di autocorrelazione sono normalizzati come in (A.9), allora l'errore minimo E_i è diviso per $R(0)$. Si può chiamare la quantità risultante l'*errore normalizzato* V_i:

$$V_i = \frac{E_i}{R(0)} = 1 + \sum_{k=1}^{i} a_k r(k). \tag{A.11}$$

Dalla (A.10) è chiaro che

$$0 \leq V_i \leq 1, \qquad i \geq 0 \tag{A.12}$$

Quindi, dalla (A.8e) e dalla (A.11), l'errore finale normalizzato V_p è

$$V_p = \prod_{i=1}^{p} (1 - k_i^2) \tag{A.13}$$

Le quantità intermedie $k_i, 1 \leq i \leq p$, sono conosciute come i *coefficienti di riflessione*. Questi valori servono a fornire un'indicazione del comportamento del predittore, e possono essere utilizzati per migliorarne l'efficienza. Spesso e a seconda dei casi è necessario stabilizzare i coefficienti tramite procedimenti opportuni [36], ma essenzialmente il metodo di previsione lineare è riassunto dalla teoria appena descritta.

A.2 Grid Harvest Service

A.2.1 Tempo di completamento di un sotto-task sulla singola macchina

Data una macchina inattiva quando un task parallelo arriva su questa, il tempo di completamento parallelo T_k, sulla macchina k può essere espresso come:

$$T_k = X_1 + Y_1 + X_2 + Y_2 + \ldots + X_S + Y_S + Z \tag{A.14}$$

S è il numero totale di interruzioni che occorrono durante il processamento di un task parallelo dovuto all'arrivo di uno o più lavori sequenziali. $X_i, Y_i, i = 1, \ldots, S$, rappresentano rispettivamente il tempo di computazione consumato dal task parallelo e quello consumato dai lavori sequenziali. Z è il tempo di esecuzione dell'ultimo processo che finisce il task parallelo. Si ha

$$X_1 + X_2 + \ldots + X_s + Z = w \tag{A.15}$$

e quindi

$$T_k = w + Y_1 + Y_2 + \ldots + Y_S \tag{A.16}$$

Sia $\Gamma(0) = 0$ e $\Gamma(S) = X_1 + X_2 + \ldots + X_S$, per $S > 0$. Allora, dalla (A.15), si ha la variabile aleatoria $S \in \{0, 1, \ldots, \infty\}$ e $\Gamma(S) < w$, $\Gamma(S+1) \geq w$.

Si ricordi che l'arrivo dei lavori locali segue una distribuzione di Poisson mentre X_i è un variabile aleatoria distribuita esponenzialmente. $\Gamma(s)$ è quindi una variabile aleatoria distribuita tramite una distribuzione Gamma per una data $S = s > 0$.

Si assuma che il processo locale sulla macchina può essere trattato come un sistema a code *M/G/1*. Un sistema M/G/1 rappresenta un server singolo che gestisce una coda di processi il cui arrivo è rappresentato da una distribuzione di Poisson con tasso d'arrivo medio pari a λ e tasso di servizio pari a μ. I tempi di servizio dei lavori sono indipendenti e distribuiti identicamente secondo una funzione di distribuzione F_b ed una pdf f_b, i lavori sono schedulati in ordine di arrivo secondo una politica FCFS.

Allora il numero totale di interruzioni S segue una distribuzione di Poisson con parametro λ_w.

Quindi, la funzione probabilità di S soddisfa:

$$p_s = Pr(S = s) = Pr(\Gamma(s) < w, \Gamma(s+1) \geq w) = \frac{(\lambda w)^s 2^{-\lambda w}}{s!} S = s > 0 \tag{A.17}$$

Si noti che il tempo di completamento di un lavoro locale

$$T_k = w + Y_1 + Y_2 + \ldots + Y_s, \tag{A.18}$$

dove $Y_j, j = 1, 2, \ldots s$, sono variabili casuali indipendenti ed identicamente distribuite rappresentanti il $j$mo periodo in cui la macchina è occupata dai lavori locali. Sotto l'assunzione che la schedulazione dei lavori locali segua un sistema di code $M/G/1$, si ricavano il valore atteso primo e secondo di Y_j:

$$E(Y_j) = \frac{1}{\mu - \lambda} \tag{A.19}$$

$$E(Y_j^2) = \frac{\mu(\mu^2\sigma^2 + 1)}{(\mu - \lambda)^3} \tag{A.20}$$

La media e la varianza di T_k possono essere ottenute attraverso la seguente espressione:

$$\begin{aligned} E(T_k) &= E(E(T_k|S)) = E(w + Y_1 + Y_2 + \ldots + Y_s|S) \\ &= E(w + SE(Y_1)) = w(1 + \lambda E(Y_1)) \\ &= \tfrac{1}{1-\rho} w \end{aligned} \tag{A.21}$$

$$\begin{aligned} V(T_k) &= E(V(T_k|S)) + V(E(T_k|S)) \\ &= E(V(w + U(S)|S)) + V(E(w + U(S)|S) \\ &= E(SV(Y_1)) + V(w + SE(Y_1)) \\ &= \lambda w V(Y_1) + \lambda w E^2(Y_1) = \lambda w E(Y_1^2) \\ &= \lambda w \tfrac{\mu(\mu^2\sigma^2+1)}{(\mu-\lambda)^3} = \tfrac{\rho}{(1-\rho)^3} \tfrac{(\theta^2+1)}{\mu} w \end{aligned} \tag{A.22}$$

dove $\rho = \lambda/\mu$ è il tasso di utilizzazione della macchina, mentre $\theta = \sigma\mu$ è il coefficiente di variazione del servizio.

Si semplifichi ritenendo che sulla macchina il sistema sia di tipo *M/M/1*. Un sistema M/M/1 è un caso particolare del M/G/1 dove la distribuzione dei tempi di servizio dei lavori è data da una distribuzione esponenziale con parametro μ. Si ha allora che $\sigma = \mu, \theta = 1$ e

$$V(T_k) = \frac{2\rho}{(1-\rho)^3\mu} w \tag{A.23}$$

Dalla (A.21) e dalla (A.22) si possono fare le seguenti riflessioni riguardanti il tempo di completamento dei sottotask paralleli:

- La media e la varianza del tempo di completamento del sotto-task sono proporzionali al workload del sotto-task. Quindi il compito principale nello stimare il tempo di completamento del task parallelo è analizzare l'influenza dell'utilizzazione del proprietario della macchina.

- La media dei tempi di completamento di un sotto-task parallelo è indipendente dalla variazione del tempo di servizio ed è reciproca dell'utilizzazione della macchina.

- Dalla (A.21) e dalla (A.22), si può trovare facilmente il coefficiente di variazione del tempo di completamento di un sotto-task $[V(T_k)/E(T_k)^2]$, che tende a 0 al crescere del tempo del task parallelo. Il coefficiente di variazione è inoltre positivo relativamente all'utilizzazione per le singole macchine. Da qui si deduce che l'incremento d'utilizzo della macchina o la variabilità nel servire i lavori locali causa una maggiore variabilità nel tempo di completamento dei sotto-task paralleli.

A.2.2 Tempo di completamento del task parallelo

Si assuma che le macchine utilizzate per il calcolo parallelo siano m, allora T può essere espresso come:

$$T = Max\{T_k, k = 1, 2, \ldots, m\} \tag{A.24}$$

Assumendo che gli utilizzi delle differenti macchine siano indipendenti, la probabilità che i task paralleli finiscano al tempo t è uguale a

$$Pr(T \leq t) = Pr(Max_{1\leq k\leq m}\{T_k, k = 1, \ldots, m\} \leq t) = \prod_{k=1}^{m} Pr(T_k \leq t) \tag{A.25}$$

Dalla (A.16),

$$T_k = w_k + Y_{k1} + Y_{k2} + \ldots + Y_{kS_k} = w_k + U(S_k), \tag{A.26}$$

dove Y_k è il $j$mo periodo in cui la macchina k è occupata,

$$U(S_k) = \begin{cases} 0, & \text{se } S_k = 0 \\ Y_{k1} + Y_{k2} + \ldots + Y_{kS_k}, & \text{se } S_k > 0 \end{cases} \tag{A.27}$$

Come precedentemente definito, si conosce che Y_i è il periodo occupato della variabile aleatoria rappresentante l'occupazione del sistema dovuta ai lavori locali e S_k è una variabile aleatoria Poissoniana con tasso λw.

Si noti che $Pr(S_k = 0) = e^{-\lambda w_k}$. La distribuzione di T_k è una combinazione delle variabili aleatorie.

$$\begin{aligned} Pr(T_k \leq t) &= Pr(T_k \leq t|S_k = 0)Pr(S_k = 0) \\ &\quad + Pr(T_k \leq t|S_k > 0)Pr(S_k > 0) \\ &= \begin{cases} e^{-\lambda w_k} + (1 - e^{-\lambda w_k})Pr(U(S_k) \leq t - w_k|S_k > 0 & \text{se } t \geq w_k \\ 0, & \text{se } t < w_k \end{cases} \end{aligned} \tag{A.28}$$

Si deve ora trovare la distribuzione di $U(S_k|S_k > 0)$.

Conseguentemente, la probabilità che il processo parallelo finisca con il tempo t è

$$Pr(T \leq t) = \begin{cases} \prod_{k=1}^{m}[e^{-\lambda w_k} + (1 - e^{-\lambda w_k})Pr(U(S_k) \leq t - w_k | S_k > 0)], & \text{se } t \geq w_{max} \\ 0, & \text{altrimenti,} \end{cases} \tag{A.29}$$

dove $w_{max} = Max\{w_k\}$. Nel caso particolare che la macchina abbia un utilizzo distribuito in maniera uniforme ed un tasso di carico distribuito equalmente w, allora si ha

$$Pr(T \leq t) = \begin{cases} [e^{-\lambda w} + (1 - e^{-\lambda w})Pr(U(S_1) \leq \tau | S_1 > 0)]^m, & \text{se } \tau > 0 \\ 0, & \text{altrimenti} \end{cases} \tag{A.30}$$

dove $\tau = t - w$. Se la distribuzione di probabilità $Pr(U(S_k) \leq u | S_k > 0)$ può essere identificata, la distribuzione del tempo di completamento del task parallelo $Pr(T \leq t)$ può essere calcolata. La media e la varianza del tempo di completamento del task parallelo possono essere anch'esse calcolate. In ogni caso, è difficile se non impossibile, ottenere un espressione esplicita di $Pr(U(S_k) \leq u | S_k > 0)$ basandosi sui risultati esistenti in probabilità. É possibile però approssimare questo valore se si conosce la media e la deviazione standard della variabile aleatoria e se si può approssimare la funzione di distribuzione tramite una conosciuta in letteratura.

Tramite l'uso delle seguenti equazioni:

$$\begin{aligned} E(T_k) &= E(T_k|S_k > 0)Pr(S_k > 0) + E(T_k|S_k = 0)Pr(S_k = 0) \\ V(T_k) &= V(T_k|S_k > 0)Pr(S_k > 0) + V(T_k|S_k = 0)Pr(S_k = 0) \end{aligned} \tag{A.31}$$

e dai risultati $E(T_k = \frac{1}{1-\rho}w$, $Pr(S = 0) = e^{-\lambda w}$, $E(T_k|S = 0) = w$, e $V(T_k|S = 0) = 0$, si ha

$$\begin{aligned} E(T_k|S_k > 0) &= \tfrac{1}{1-e^{-\lambda w}}\left[\tfrac{1}{1-\rho}w - we^{-\lambda w}\right], \\ &= w + \tfrac{1}{1-e^{-\lambda w}}\tfrac{\rho}{1-\rho}w \\ V(T_k|S_k > 0) &= \tfrac{1}{1-e^{-\lambda w}}V(T_k) \\ &= \tfrac{1}{1-e^{-\lambda w}}\tfrac{\rho}{(1-\rho)^3}\tfrac{(\theta^2+1)}{\mu}w \end{aligned} \tag{A.32}$$

Quindi,

$$E(U(S_k)|S_k > 0) = E(T_k|S_k > 0) - w = \frac{1}{1 - e^{-\lambda w}}\frac{\rho w}{1 - \rho} \tag{A.33}$$

e

$$\begin{aligned} V(U(S_k)|S_k > 0) &= V(T_k|S_k > 0) \\ &= \tfrac{1}{1-e^{-\lambda w}}\tfrac{\rho}{(1-\rho)^3}\tfrac{(\theta^2+1)}{\mu}w \end{aligned} \tag{A.34}$$

Non rimane che trovare un'approssimazione possibile della distribuzione di funzione per $U(S_k)$ dato $S_k > 0$.

I creatori di GHS hanno poi proceduto a ricavare sperimentalmente da alcune simulazioni, quale fosse la distribuzione che meglio rappresenta $U(S_k)$, considerando cinque tempi differenti di distribuzione dei servizi dei task locali, utilizzando per essi una distribuzione esponenziale, una distribuzione Erlang, una distribuzione Gamma, una Log-Normale ed una distribuzione Normale troncata (in cui non sono ammessi i valori estremi per le code).

Dai risultati degli esperimenti simulati si è ottenuto che le distribuzioni Gamma, Lognormale e Weibull sono quelle che meglio si adattano ad approssimare la distribuzione di $U(S)|S > 0$, si è rilevato inoltre che l'efficienza tra l'uso di una o l'altra è minima. La distribuzione Gamma risulta essere l'approssimazione migliore se l'utilizzazione del sistema è bassa (ossia pari al 5-15 per cento) e se la domanda dei sotto-task paralleli è ragionevolmente lunga. La Weibull invece si mostra come la distribuzione che si adatta meglio nel caso in cui l'utilizzazione della macchina locale è media (ossia maggiore uguale al 50 per cento).

Altro risultato è stato mostrare come il partizionamento migliore di un task parallelo per un sistema omogeneo, corrisponde a quello in cui si viene ad avere una distribuzione del carico uguale su tutte le macchine. Questo risultato non è valido all'interno di sistemi eterogenei. Infatti in tali sistemi uno dei problemi principali è come partizionare i task paralleli e allocare i sotto-task sulle macchine al fine di ottenere le prestazioni migliori.

Si è proceduto a determinare allora una *regola di partizionamento* per ogni task parallelo. Dato un sotto-task parallelo con un carico richiesto w_k, dal modello matematico si è mostrato che questo possiede una media ed una deviazione standard $E(T_k), V(T_k)$ per il tempo di completamento del sotto task su ciascuna macchina k. Dalle sperimentazioni [26] si è sottolineato come la media del tempo di completamento dei sotto-task ed il tempo di utilizzo della macchina abbiano un influenza rilevante sul tempo di completamento finale parallelo rispetto alla variazione della distribuzione del servizio.

Gli autori hanno deciso allora di partizionare il task parallelo W in sotto-task con carico w_k per le macchine k in maniera tale da avere la stessa media dei tempi di completamento di ciascun sotto-task sui differenti calcolatori. Gli autori hanno battezzato questo approccio come *mean time balancing partition*. Osservando la (A.21), il tempo di completamento atteso del sotto-task è $\frac{w_k}{1-\rho_k}$. Il carico del sotto-task w_k per la $k$ma macchina è determinato dall'equazione

$$\frac{w_k}{1-\rho_k} = a \qquad oppure \qquad w_k = a(1-\rho_k) \tag{A.35}$$

Si osservi che il lavoro richiesto totale per il task parallelo è W che implica $a = W/(m - \sum_{k=1}^{m} \rho_k)$. Quindi si ha la

$$w_k = \frac{W(1-\rho_k)}{m - \sum_{k=1}^{m} \rho_k} = \frac{W}{m} \frac{1-\rho_k}{1-\bar{\rho}} \tag{A.36}$$

dove $\bar{\rho} = \frac{1}{m} \sum_{k=1}^{m} \rho_k$, è l'utilizzazione media del sistema. Sono facilmente ricavabili i valori attesi per $E(T_k)$ e $V(T_k)$.

A.3 Network Weather Service

A.3.1 Predittore ADAPT_AVG(t)

Sia $K(t)$ il valore di K al tempo t, e sia

$$err_i(t) = (value(t) - SW\_AVG(t, K(t) + i))^2 \tag{A.37}$$

. Allora si definisce

$$ADAPT\_AVG(t) = \begin{cases} SW\_AVG(t, K(t) - 1) & \text{if } min_{i=-1,0,1} err_i(t) = err_{-1}(t) \\ SW\_AVG(t, K(t)) & \text{if } min_{i=-1,0,1} err_i(t) = err_0(t) \\ SW\_AVG(t, K(t) + 1) & \text{if } min_{i=-1,0,1} err_i(t) = err_1(t) \end{cases} \tag{A.38}$$

e

$$K(t+1) = \begin{cases} K(t) - 1 & \text{if } min_{i=-1,0,1} err_i(t) = err_{-1}(t) \\ K(t) & \text{if } min_{i=-1,0,1} err_i(t) = err_0(t) \\ K(t) + 1 & \text{if } min_{i=-1,0,1} err_i(t) = err_1(t) \end{cases} \tag{A.39}$$

Al fine di minimizzare l'errore, ad ogni passo si corregge il valore di K facendo uso dell'errore quadratico $err_i(t)$ o tramite una misurazione dell'errore assoluto percentuale. Si noti che il valore di $K(t)$ dev'essere incluso come parte della storia per far funzionare correttamente $ADAPT\_AVG$, e che $K(0)$ è specificato a qualche valore di partenza ragionevole. Si restringe arbitrariamente anche K in maniera tale che sia compreso tra il valor minimo ed il valor massimo. Specificare una soglia massima limita la complessità computazionale del predittore mentre una soglia minima evita di rimanere "bloccati" in un minimo locale.

A.3.2 Predittore SW_AVG(t)

Come con SW_AVG la scelta di K può essere difficile da determinare. Si include quindi un filtro mediano adattativo che è analogo all'$ADAPT\_AVG$

$$ADAPT\_MED(t) = \begin{cases} MEDIAN(t, K(t)-1) & \text{se } min_{i=-1,0,1} err_i(t) = err_{-1}(t) \\ MEDIAN(t, K(t)) & \text{se } min_{i=-1,0,1} err_i(t) = err_0(t) \\ MEDIAN(t, K(t)+1) & \text{se } min_{i=-1,0,1} err_i(t) = err_1(t) \end{cases} \tag{A.40}$$

dove $K(t)$ è il valore al tempo t e

$$err_i(t) = (value(t) - MEDIAN(t, K(t)+i))^2 \tag{A.41}$$

dove $K(t+1)$ è determinato dall'Equazione (A.39).

I filtri basati sulla mediana sono d'interesse perché non soffrono degli effetti causati da rapide deviazioni ed impulsi durante la creazione di una previsione. In compenso sfumano meno rispetto ai metodi basati su media, fornendo previsioni con un quantitativo considerevole di variazioni brusche ed impreviste.

É possibile combinare i vantaggi positivi di entrambe le classi di metodi tramite filtri *trimmed mean* che calcolano la media dei valori centrali $K - 2 * \alpha * K$ tramite una finestra mobile di dimensione K per $(0 < \alpha < 0,5)$ Si definisce

$$T = \lfloor \alpha * K \rfloor \tag{A.42}$$

per una finestra di dimensione K, la *trimmed mean* è

$$TRIM\_MEAN(t, K, T) = \sum_{j=T+1}^{K-T+1} \frac{1}{K - 2 * T} Sort_K(j) \tag{A.43}$$

Bibliografia

[1] *OurGrid: An approach to easily assemble grids with equitable resource sharing*, June 2003.

[2] *BOINC: a system for public-resource computing and storage*, 2004.

[3] J. H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. *ipdps*, 14:238b, 2004.

[4] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[5] Nazareno Andrade, Francisco Brasileiro, Walfredo Cirne, and Miranda Mowbray. Discouraging free riding in a peer-to-peer cpu-sharing grid. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 129–137. IEEE Computer Society, 2004.

[6] Cosimo Anglano and Massimo Canonico. Fault-tolerant scheduling for bag-of-tasks grid applications. In *EGC*, pages 630–639, 2005.

[7] S. Asmussen, O. Nerman, and M.Olsson. Fitting phase-type distributions via the em algorithm. *Scandivian Journal of Statistics*, 23:419–441, 1996.

[8] P. Barham. Xen and the art of virtualization, 2003.

[9] J. Brevik, D. Nurmi, and R. Wolski. Quantifying machine availability in networked and desktop grid systems, 2003.

[10] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid, 2000.

[11] Massimo Canonico. *Scheduling Algorithms for Bag-of-Tasks Applications on Fault-Prone Desktop Grids*. PhD thesis, Università degli Studi di Torino, 2002-2005.

[12] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov, and Francine Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Proc. 9th Heterogeneous Computing Workshop (HCW)*, pages 349–363, Cancun, Mexico, May 2000.

[13] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988.

[14] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system, 2003.

[15] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauve, Fabricio A. B. Silva, Carla O. Barros, and Cirano Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. *icpp*, 00:407, 2003.

[16] Thomas A. DeFanti, Ian Foster, Michael E. Papka, Rick Stevens, and Tim Kuhfuss. Overview of the I-WAY: Wide-area visual supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3):123–131, Summer/Fall 1996.

[17] J. Durbin. The fitting of time series models. *RevueInstStat*, 28:233–243, 1960.

[18] A. Quarteroni e F. Saleri. *Introduzione al Calcolo Scientifico*. Springer, 2nd edition, 2004.

[19] Dietmar W. Erwin and David F. Snelling. UNICORE: A Grid computing environment. *Lecture Notes in Computer Science*, 2150:825–??, 2001.

[20] Gilles Fedak, Cecile Germain, Vincent Neri, and Franck Cappello. Xtremweb: A generic global computing system. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 582, Washington, DC, USA, 2001. IEEE Computer Society.

[21] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[22] Ian Foster. Globus toolkit version 4: Software for service-oriented systems, 2006.

[23] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.

[24] Ian T. Foster and Carl Kesselman. Computational grids. In *VECPAR*, pages 3–37, 2000.

[25] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 55, Washington, DC, USA, 2001. IEEE Computer Society.

[26] Linguo Gong, Xian-He Sun, and Edward F. Watson. Performance modeling and prediction of nondedicated network computing. *IEEE Trans. Comput.*, 51(9):1041–1055, 2002.

[27] Peter Gradwell. Overview of grid scheduling systems, 2002m.

[28] Francois Grey, Matti Heikkurinen, Rosy Mondardini, and Robindra Prabhu. Cern grid café, 2006.

[29] Andrew S. Grimshaw, Wm. A. Wulf, and CORPORATE The Legion Team. The legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1):39–45, 1997.

[30] Robert L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

[31] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. A framework for adaptive execution in grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.

[32] N. Levinson. The wiener rms error criterion in filter design and prediction. 25, 1947.

[33] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[34] Pangfeng Liu and Jan-Jan Wu. Optimal replica placement strategy for hierarchical data grid systems. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 417–420, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.

[36] J. Makhoul. Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63(4):561–580, 1975.

[37] James Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems.

[38] W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[39] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[40] E. A. Robinson. *Statistical Communication and Detection*. Griffin, London, 1967. — Sig.: Bk49.

[41] Luis F. G. Sarmenta. Volunteer computing.

[42] Xiang Song and Raj Kumar. Gridlite: An infrastructure for provisioning and managing grid services in resource constrained devices. In *GCA*, pages 127–133, 2006.

[43] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[44] J. Weissman and D. Womack. Fault tolerant scheduling in distributed networks, 1996.

[45] Rich Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998.

[46] Rich Wolski, Neil Spring, and Chris Peterson. Implementing a performance forecasting system for metacomputing: the network weather service. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–19, New York, NY, USA, 1997. ACM Press.

[47] Ming Wu and Xian-He Sun. Grid harvest service: a performance system of grid computing. *J. Parallel Distrib. Comput.*, 66(10):1322–1337, 2006.

[48] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Software - Practice and Experience*, 23(12):1305–1336, 1993.