Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il modello di computing è radicalmente cambiato negli ultimi anni. Sono infatti sempre di più i centri di calcolo, sia nel mondo industriale che scientifico, in cui il singolo computer centrale di tipo mainframe, a cui tutti gli utenti accedevano attraverso terminali remoti, è stato sostituito con workstation, collegate tra loro attraverso una rete locale (*LAN: Local Area Network*). Grazie infatti ai notevoli sviluppi tecnologici, è ora possibile acquistare workstation con un rapporto prestazioni/prezzo nettamente superiore a quello dei supercalcolatori di tipo mainframe [8] [9] [22] [30]. Anche l'impiego di personal computer è sempre più diffuso, visto che sono in grado di fornire elevate prestazioni a costi molto competitivi.

L'utilizzo di un sistema di calcolo distribuito offre altri vantaggi, oltre alla semplice convenienza economica: rispetto ai sistemi costituiti da un unico calcolatore, è infatti più facile gestire variazioni della "dimensione" dei problemi da trattare, è più semplice configurare il sistema secondo le proprie specifiche esigenze, ecc... Se poi si considera un ambiente eterogeneo, dove cioè i calcolatori possono essere di diversa architettura, è possibile soddisfare in modo più efficiente le diverse richieste computazionali. Non esiste infatti, e probabilmente non esisterà nemmeno in futuro, un singolo modello di calcolatore adatto a risolvere tutti i tipi di problemi: certe architetture si prestano meglio di altre ad eseguire particolari tipi di codice (per fare un esempio, le architetture SIMD sono più adatte per le computazioni matriciali, mentre le architetture MIMD sono particolarmente indicate per applicazioni costituite da moduli con una limitata intercomunicazione [27]) [24].

Vi sono però varie difficoltà e problematiche, ancora non del tutto risolte, che devono essere considerate in un ambiente di calcolo distribuito ed eterogeneo: è necessario trattare opportunamente le differenze tra i sistemi considerati (incompatibilità binaria, diversa rappresentazione interna dei dati, ecc...), il programmatore deve gestire direttamente diversi dettagli implementativi (la sincronizzazione e la comunicazione tra i vari moduli del programma, il mapping dei processi ai processori disponibili, ecc...), sono necessari adeguati strumenti per il *code profiling* (il procedimento che permette di fornire una stima del tempo di esecuzione delle varie applicazioni sui diversi calcolatori), per il *benchmarking*, per il *mapping* (l'assegnazione dei processi ai processori disponibili), ecc... Inoltre l'attività di amministrazione di un ambiente costituito da decine, o centinaia, o migliaia di workstation può essere molto complessa e richiedere molte risorse.

Una delle realtà in cui il modello di calcolo è cambiato, e in cui il problema della gestione efficiente di un sistema di calcolo distribuito ed eterogeneo è molto sentito, è l'ambiente della fisica delle alte energie (*HEP: High Energy Physics*). La ricerca fisica di tipo teorico e di tipo sperimentale è infatti fortemente vincolata dalle risorse computazionali disponibili limitate: se non vi fossero questi limiti potrebbe essere fatta ricerca ad un diverso livello. È quindi di fondamentale importanza poter gestire nel miglior modo possibile le risorse di calcolo disponibili.

Un tipico programma di elaborazione che deve essere considerato in un esperimento di fisica delle alte energie si compone di unità computazionali elementari, chiamate *task*, ognuna delle quali gestisce un particolare aspetto del problema. I vari task vengono poi combinati tra loro, costituendo le cosiddette *production*. Solitamente una production è costituita da una serie di task posti in "cascata" l'uno con l'altro, ma a volte è necessario considerare strutture più complesse. Le production vengono in genere utilizzate per elaborare un numero di dati molto elevato.

Attualmente non esiste un modello unico per la gestione di queste elaborazioni. Esistono specifici sistemi, che sono però strettamente vincolati al tipo di job da eseguire e alla configurazione dell'ambiente di calcolo considerato. Questi sistemi presentano poi vari problemi, quali il livello di automatizzazione piuttosto limitato (è richiesto un controllo e una gestione praticamente costante da parte dell'utente), un utilizzo spesso non ottimale delle risorse di calcolo disponibili, ecc...

Per gli esperimenti della prossima generazione (primi fra tutti gli esperimenti LHC al Cern) i problemi saranno ancora più complessi, poiché il volume dei dati da gestire sarà molto superiore a quello degli esperimenti attuali.

In questa tesi viene proposto un sistema, chiamato DEMI (Distributed hEterogeneous coMputing envIronment) per la programmazione e la gestione di elaborazioni in un ambiente di calcolo distribuito ed eterogeneo. Attraverso questo sistema l'utente ha la possibilità di definire propri programmi (production), costituiti da un insieme di processi sequenziali (task). I vari task possono essere codificati usando linguaggi di programmazione diversi, e ognuno di essi può essere implementato anche per più piattaforme.

I task di una produzione possono essere eseguiti in maniera concorrente, rispettando però le relazioni di dipendenza sui dati. Un task, infatti, può aver bisogno per l'elaborazione di dati prodotti da altri task: finché l'esecuzione di questi task non viene completata, e quindi finché non sono stati prodotti i relativi output, il task considerato deve rimanere in stato di attesa.

Le produzioni possono essere rappresentate attraverso grafi "data-flow" di tipo aciclico (*DAG: Directed Acyclic Graph*), in cui i nodi rappresentano i processi (task), mentre gli archi rappresentano i vincoli di dipendenza sui dati. La regola di firing associata a ciascun nodo del grafo, cioè la condizione che definisce quando l'esecuzione del processo associato al nodo può cominciare, è *data-driven*: l'elaborazione può iniziare quando "arrivano" al nodo i dati, ovvero quando tutti gli input del task sono disponibili.

Nel formalismo grafico adottato per la rappresentazione delle produzioni, gli archi del grafo intersecano dei "box", che rappresentano i dati (sotto forma di file) prodotti dai task, in attesa di essere usati come input da altri task.

L'ambiente di calcolo che deve essere considerato è composto da un insieme di siti, distribuiti geograficamente, connessi tra loro attraverso una rete geografica. Ogni sito rappresenta un metasistema: un sistema di elaborazione distribuito, costituito da calcolatori indipendenti ed eterogenei, collegati tra loro attraverso una rete locale. La gestione di ogni sito è affidata ad un amministratore locale, che in particolare deve configurare e gestire le risorse locali, mentre un amministratore globale è responsabile delle operazioni che riguardano il sistema complessivo.

Anche se si fa riferimento alle elaborazioni tipiche di un esperimento di fisica delle alte energie, quanto detto può essere applicato ad altre realtà in cui si devono affrontare problemi analoghi.

La tesi presenta lo studio e l'analisi del problema, e la progettazione e implementazione di un primo prototipo del sistema.

Come metodologia di sviluppo è stato adottato il ciclo di vita a cascata: dopo un'analisi preliminare del problema, sono stati definiti in maniera dettagliata i requisiti da considerare. È stata poi realizzata la progettazione del sistema, definendo i vari moduli e le relative funzionalità.

Per gestire in maniera persistente e in modo efficiente le varie informazioni (riguardanti l'ambiente di calcolo, le configurazioni delle elaborazioni, lo stato dell'esecuzione) è stata definita un'apposita base di dati, per la cui progettazione concettuale si è utilizzato il modello a oggetti. Lo schema ottenuto è stato poi trasformato in uno schema logico relazione, per permettere l'implementazione della base di dati in un sistema di questo tipo. La base di dati è stata organizzata gerarchicamente in una parte globale, per le informazioni che riguardano il sistema complessivo e, in ogni sito, in una parte locale, per i dati specifici del sito considerato.

È stato implementato un primo prototipo del sistema: in questa prima fase si è considerato un ambiente costituito da un singolo sito. Il sistema è stato integrato con un *load sharing system*, in grado di gestire in maniera efficiente l'esecuzione di job in un ambiente di calcolo distribuito. Infine si è verificata la funzionalità del sistema, considerando diversi tipi di test.

Nel capitolo 2 viene descritto l'ambiente della fisica delle alte energie, considerando quali sono i problemi computazionali che si devono gestire, con riferimento sia agli esperimenti attuali, che a quelli della prossima generazione. Nel capitolo 3 viene introdotto il sistema DEMI, le cui funzionalità e caratteristiche sono descritte nel capitolo 4. Nel capitolo 5 vengono presentati alcuni sistemi di load sharing, che permettono di gestire in maniera ottimale le elaborazioni in un ambiente di calcolo distribuito. I capitoli 6 e 7 descrivono la progettazione del sistema DEMI: nel capitolo 6 viene presentato il progetto della base di dati utilizzata per la gestione delle varie informazioni, mentre nel capitolo 7 viene descritta l'architettura software del sistema. Nel capitolo 8 si descrive la fase di implementazione, e si presentano esempi di utilizzo del sistema DEMI. Il capitolo 9 conclude la tesi.

Capitolo 2

L'ambiente HEP

L'obiettivo di un generico esperimento di fisica delle alte energie è quello di scoprire quali sono le particelle costituenti la materia, e le forze che esistono tra esse. In questi studi è necessario elaborare un numero molto elevato di dati, in parte acquisiti dagli apparati sperimentali, e in parte generati attraverso opportuni programmi di simulazione.

In questo capitolo, dopo una breve introduzione sugli esperimenti HEP, si analizzano quali sono le caratteristiche e i requisiti per le elaborazioni che devono essere gestite. Oltre a considerare gli esperimenti attuali, si fa riferimento anche agli esperimenti della prossima generazione, in cui il volume dei dati da gestire sarà dell'ordine dei Pbyte: per questi esperimenti i problemi computazionali da gestire saranno quindi molto più complessi.

2.1 Gli esperimenti HEP

In un generico esperimento di fisica delle alte energie (*HEP: High Energy Physics*) vengono fatti collidere fasci di particelle con bersagli fissi, o con particelle di altri fasci. Studiando i frammenti che vengono prodotti in queste collisioni, è possibile scoprire quali sono i costituenti della materia, e le forze che operano tra questi elementi [20]. A tal fine si utilizzano appositi *detector*, costituiti da più "strati" tra i quali passano le particelle: attraverso questi apparati è possibile identificare tali particelle, ricostruire le loro tracce e calcolare i loro parametri fisici (masse, energie, momenti, ecc...). Ogni detector è in grado di fornire solo specifiche informazioni, e può identificare solo particolari tipi di particelle o interazioni: per questo motivo si devono considerare più tipi di detector [49].

Un grande esperimento HEP di solito coinvolge una collaborazione che consiste in centinaia o migliaia di fisici, appartenenti a università e istituti di tutto il mondo. Il detector e gli altri apparati sperimentali relativi ad un esperimento HEP sono installati in uno specifico laboratorio, in genere chiamato il *sito principale* dell'esperimento.

In tutti gli esperimenti di fisica delle alte energie l'unità di informazione è l'**evento**: fondamentalmente un evento è una "registrazione", in forma digitale, delle tracce lasciate dalle particelle dopo una collisione [49].

I vari eventi sono indipendenti tra loro: ogni evento rappresenta un'entità a sé stante, e non è in nessun modo correlato agli altri eventi. L'elaborazione che viene fatta su un evento è quindi del tutto indipendente dall'elaborazione che viene eseguita sugli altri eventi.

Poiché la ricerca, la verifica e lo studio delle varie teorie fisiche viene fatta utilizzando metodologie di tipo statistico, è fondamentale poter disporre di un numero molto elevato di eventi da poter elaborare. In un generico esperimento HEP dell'ultima generazione vengono generalmente acquisiti dagli apparati sperimentali ogni anno un numero di eventi che può variare tra 10^6 e 10^{12}. Ogni evento ha una dimensione che di solito varia tra 1 Kbyte e 1 Mbyte, a seconda del tipo di analisi fisica che deve essere fatta, e a seconda del tipo di detector utilizzato. Per quel che riguarda i requisiti in termini di risorse computazionali, per elaborare un evento è necessario un tempo di calcolo che può variare tra qualche millisecondo fino a centinaia di secondi, a seconda del tipo di analisi considerata e, ovviamente, a seconda del tipo di processore utilizzato.

Questi numeri relativi ai "volumi" di dati che devono essere considerati in un esperimento HEP, cresceranno approssimativamente per un ulteriore ordine di grandezza nei prossimi anni: in esperimenti come CDF, D0, LEP, HERA e le B factory a SLAC e a KEK si acquisiranno dagli apparati sperimentali ogni anno una quantità di dati dell'ordine delle decine o centinaia di Tbyte. Si stima che con i collider dell'esperimento LHC al Cern, a partire dalla metà del prossimo decennio, verranno acquisiti ogni anno dati dell'ordine dei Pbyte [11]. In tabella 2.1 vengono mostrati i volumi di dati che saranno acquisiti ogni anno per alcuni esperimenti HEP [56].

Per poter elaborare tutti questi dati è necessario poter disporre di una enorme potenza di calcolo. L'ambiente HEP "lavora" sempre ai limiti della tecnologia informatica: lo studio e il progetto di un esperimento di fisica delle alte energie deve necessariamente considerare quali sono i vincoli posti dalle risorse di calcolo disponibili.

È quindi chiaro come sia di fondamentale importanza poter gestire nel modo più efficiente possibile queste risorse computazionali.

Esperimento	Laboratorio	Inizio acquisizione dati	Raw Data da acquisire
BELLE	KEK Tsukuba (Giappone)	1998	30 Tbyte / anno
NA45	CERN Ginevra (Svizzera)	1999	30 - 50 Tbyte / anno
BaBar	SLAC San Francisco (USA)	1999	200 Tbyte / anno
Esperimenti RHIC (PHENIX, STAR)	BNL Long Island (USA)	1999	200 - 300 Tbyte / anno
COMPASS	CERN Ginevra (Svizzera)	1999	300 Tbyte / anno
D0	FNAL Chicago (USA)	2000	100 Tbyte / anno
CDF (RUN II)	FNAL Chicago (USA)	2000	250 Tbyte / anno
Esperimenti LHC (ATLAS, CMS, ALICE)	CERN Ginevra (Svizzera)	2005	5 Pbyte / anno

Tabella 2.1: Dati da acquisire per alcuni esperimenti HEP

2.2 Elaborazioni in HEP

Esistono molti tipi di elaborazioni che vengono fatte sugli eventi, nei vari esperimenti di fisica delle alte energie. Queste computazioni possono comunque essere divise in tre categorie principali [20]:

- **Simulazione** (o **Monte Carlo**)

  Per poter verificare le varie teorie fisiche, i ricercatori devono confrontare i *raw data* acquisiti dagli apparati sperimentali, con eventi simulati, generati attraverso specifici programmi. Di solito è necessario poter disporre di un numero di eventi simulati molto superiore al numero di eventi reali acquisiti (almeno il doppio). I dati reali (acquisiti dagli apparati sperimentali) e i dati simulati hanno la stessa struttura: in questo modo è poi possibile utilizzare gli stessi programmi per elaborare i due insiemi di dati.

  La simulazione viene anche usata per valutare l'efficienza del detector utilizzato.

- **Ricostruzione**

  In questa fase i raw data (reali e simulati) vengono elaborati, in modo da ottenere valori significativi dal punto di vista fisico. Ad esempio, per trovare la carica di una particella si ricostruisce la corrispondente traccia nel detector: analizzando poi la curvatura di questa traiettoria, dovuta alla presenza del campo magnetico, si determina appunto la carica. In questa fase si identificano anche le particelle generate nelle collisioni (una volta calcolati le corrispondenti cariche e masse).

- **Analisi**

  L'analisi può essere scomposta in varie fasi, nelle quali gli eventi reali ricostruiti e gli eventi simulati ricostruiti vengono confrontati tra loro, per verificare differenze e/o analogie.

Semplificando il problema, una tipica elaborazione può quindi essere schematizzata dal seguente pseudo-codice:

```
1  begin
2    if (necessary == true)
3      GenerateMonteCarloEvents();
4    end
5    while (necessary == true)
6          begin
7            MCRawSubset = GetMCRawSubSet();
8            RealRawSubset = GetRealRawSubSet();
9            MCReconSubset = Reconstruction(MCRawSubset);
10           RealReconSubset = Reconstruction(RealRawSubset);
11           MCOut = SpecificAnalysisProg(MCReconSubset);
12           RealOut = SpecificAnalysisProg(RealReconSubset);
13           Compare(MCOut, RealOut);
14         end
15   end
```

- Linee 2 - 4

  Se necessario (se non sono già disponibili) vengono generati eventi Monte Carlo, attraverso un opportuno programma di simulazione.

- Linee 7 - 8

  Dall'insieme costituito da tutti gli eventi reali, acquisiti dagli apparati sperimentali, si estrae un sottoinsieme di eventi: la scelta dipende dal tipo di problema fisico da considerare. Analoga operazione viene fatta per l'insieme dei raw data simulati.

- Linee 9 - 10

  Si effettua la ricostruzione (utilizzando lo stesso programma) dei due insiemi di eventi reali e simulati prima selezionati.

- Linee 11 - 12

  Utilizzando lo stesso programma di analisi, scritto per la "gestione" dello specifico problema considerato, si elaborano gli eventi "ricostruiti", reali e simulati, ottenuti nella precedente fase.

- Linea 13

  I risultati dell'analisi, sui dati reali e su quelli simulati (che solitamente sono sotto forma di distribuzioni statistiche), vengono confrontati tra loro, per verificare differenze e/o analogie.

Nella simulazione e nella ricostruzione i requisiti computazionali in termini di I/O sono modesti: l'elaborazione è "limitata" dalla CPU disponibile. Nelle fasi dell'analisi, invece, può essere necessaria anche un'elevata attività di I/O.

I programmi che vengono utilizzati per la simulazione, la ricostruzione e l'analisi sono specifici ai tipi di detector considerati, e possono essere costituiti anche da milioni di righe di codice. Alcuni di questi programmi possono essere utilizzati per molti anni; altri, invece, devono essere frequentemente modificati, quando vi è una qualsiasi modifica nei relativi detector.

In tutti questi programmi si fa uso di librerie standard di procedure, utilizzate in tutti gli esperimenti HEP (ad esempio le *librerie CERNlib*) [49]. Anche l'I/O è gestito attraverso routine standard: in questo modo non è necessaria nessuna conversione esplicita quando i dati sono prodotti da un computer, e devono poi essere utilizzati da un altro calcolatore di architettura diversa.

Quando è disponibile una nuova versione di un programma di ricostruzione/analisi, è spesso necessario usarlo per elaborare dati che erano stati già processati da una precedente release del programma. Questa procedura è chiamata *reprocessing*.

Poiché, come detto, gli eventi sono indipendenti tra loro, è possibile elaborarli in maniera autonoma e asincrona [48]. È quindi possibile "parallelizzare" le attività di simulazione, ricostruzione e analisi, utilizzando più processori. L'insieme totale degli eventi da elaborare può essere partizionato in un numero di insiemi pari al numero dei processori disponibili, e il programma che esegue l'elaborazione considerata viene "replicato" su ognuno di questi processori. In questo modo ogni processore esegue l'elaborazione considerata su un proprio insieme di dati. In questa forma di *data parallelism*, il programma considerato non è un programma parallelo: semplicemente viene eseguito in parallelo su più processori [20]. In figura 2.1 è mostrato un esempio di questo parallelismo sui dati: l'insieme dei dati di input viene partizionato in due insiemi, e due processori (*Host 1* e *Host 2*) eseguono lo stesso programma, ognuno su uno di questi due insiemi di dati.

Oltre al data parallelism, in alcuni casi si può considerare anche un'altra forma di parallelismo: il *parallelismo temporale*. In questi casi la computazione viene "scomposta" in più task, ognuno dei quali viene assegnato ad uno specifico processore [53]. In figura 2.2 è mostrato un esempio di parallelismo temporale: la computazione è composta di due task (*Task 1* e *Task 2*), che vengono associati a due processori (*Host 1* e *Host 2*).

Per questa seconda forma di parallelismo si considera una granularità di tipo *coarse-grain*, in quanto le componenti che vengono eseguite in parallelo sono applicazioni "complete", e non singole istruzioni di uno stesso programma [49].

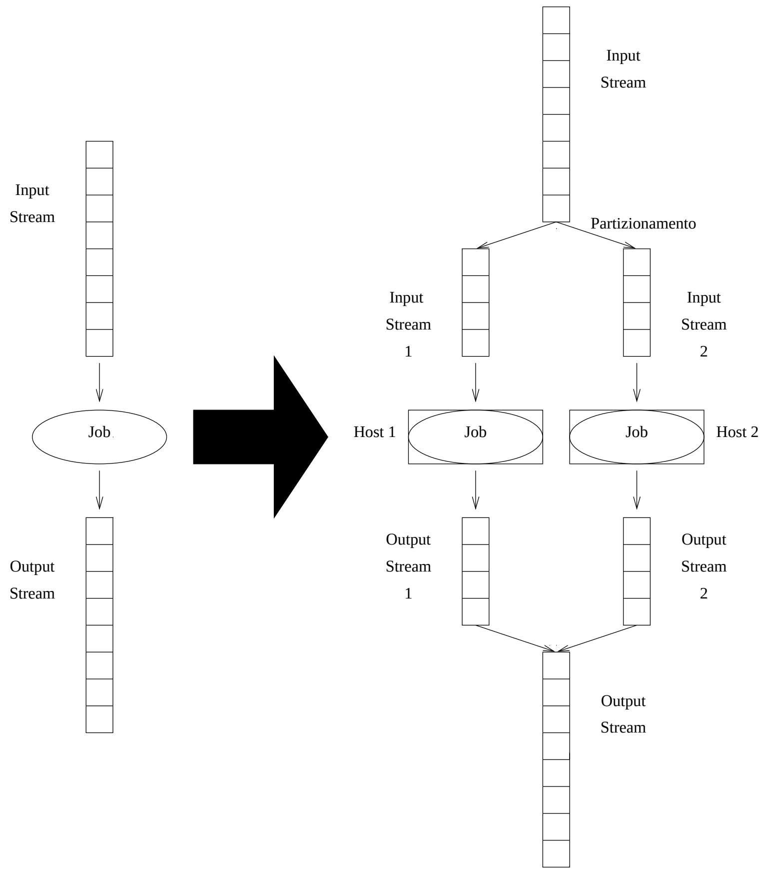


Figura 2.1: Data parallelism, usando due processori

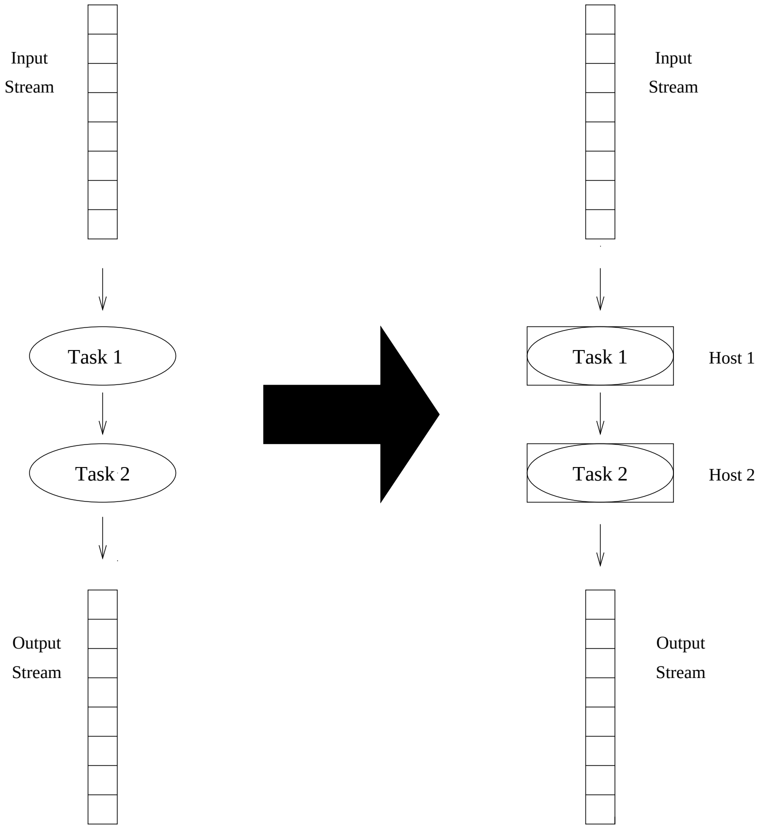


Figura 2.2: Parallelismo temporale, usando due processori

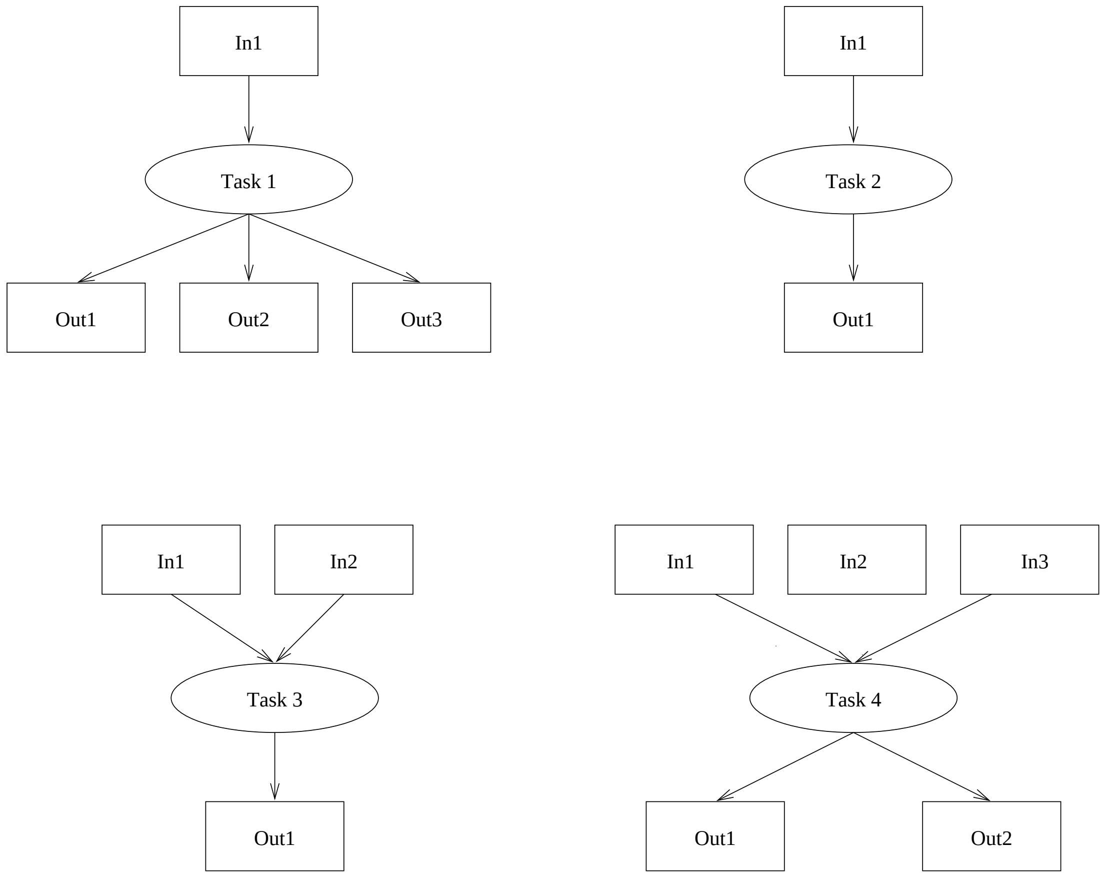


Figura 2.3: Esempi di rappresentazioni grafiche di task

2.2.1 Task e Production

Di solito un'elaborazione fisica non consiste di un singolo programma elementare, con un input e un output, ma può avere una struttura più complessa: può essere composto da più elementi computazionali elementari, che chiamiamo **task**. Ogni task rappresenta un'unità di elaborazione, responsabile di "gestire" un singolo aspetto del problema complessivo (simulazione, ricostruzione, una singola fase dell'analisi, ecc...). Un task può avere zero o più input, e zero o più output.

Graficamente si può rappresentare un task con un'ellisse, e i suoi input-output con rettangoli, come rappresentato negli esempi di figura 2.3.

I task vengono "combinati" tra loro, costituendo quella che viene chiamata **production**: una production gestisce quindi un job "completo" (ad esempio la generazione di eventi simulati, la loro ricostruzione, la loro analisi, costituita da diverse fasi). Per descrivere una production bisogna quindi specificare i task che la compongono, e le relazioni che esistono tra essi.

La struttura logica di una production può anche essere molto complessa. Ad esempio la production rappresentata in figura 2.4 è composta dai quattro task (*Task1*, *Task2*, *Task3* e *Task4*) definiti in figura 2.3.

L'input del task *Task 1* è anche l'input (*Input 1*) della production. Il primo output del task *Task 1* coincide con l'input del task *Task 2* (*Dat 2*). Gli altri due output del task *Task 1* sono input del task *Task 3* (*Dat 3* e *Dat 4*). Il secondo output del task *Task 1* è anche input del task *Task 4*. Gli output dei task *Task 2* e *Task 3* sono usati come input dal task *Task 4* (*Dat 5* e *Dat 6*). Gli output del task *Task 4* sono anche output della production (*Output 1* e *Output 2*).

I dati di input possono essere classificati in due categorie: gli *event data* e i *long term data*. Gli event data rappresentano sequenze di eventi indipendenti, che devono essere elaborati uno alla volta. I long term data, invece, sono singoli file, che rappresentano "costanti" che devono essere usate nell'elaborazione di tutti gli eventi che costituiscono gli event data. Ad esempio le *costanti di calibrazione* sono input di tipo long term.

Ogni task può avere input sia di tipo long term, che di tipo event, e produce output di tipo event [53]. Ogni task può avere zero o al più un input di tipo event, e zero o più input di tipo long term: un task non può avere due o più input di tipo event. Considerando ad esempio un task per la ricostruzione, vi sarà un input di tipo event (per la sequenza di eventi che devono essere elaborati, uno alla volta), e uno o più input di tipo long term, rappresentanti costanti da usare per l'elaborazione di tutti gli eventi.

In figura 2.5 è rappresentato un altro esempio, meno complesso, di production. In questo caso la production è costituita da due task: *Task X* e *Task Y*. Il task *Task X* non ha input e ha un output (*Output 1*), che è anche un output della production. Il task *Task Y* ha un input (*Output 1*) che è output del task *Task 1*, e un output (*Output 2*), anch'esso un output della production.

In questo caso, quindi, il risultato intermedio *Output 1* deve essere considerato un output della production (ad esempio deve essere memorizzato su nastro), e non può essere cancellato dopo essere stato usato come input dal task *Task X*. Si considera questo approccio quando è necessario gestire i reprocessing: in questi casi gli output dei task intermedi devono essere "salvati" su supporti di memoria secondaria, per poter poi eventualmente elaborare nuovamente questi dati in futuro, utilizzando nuove versioni di programmi di analisi.

Prima di eseguire un generico task di simulazione/ricostruzione/analisi, sono in genere necessarie alcune operazioni preliminari (ad esempio il load di un database di calibrazioni): queste operazioni possono richiedere per la loro esecuzione un tempo non trascurabile. Non è quindi conveniente elaborare un evento alla volta, in

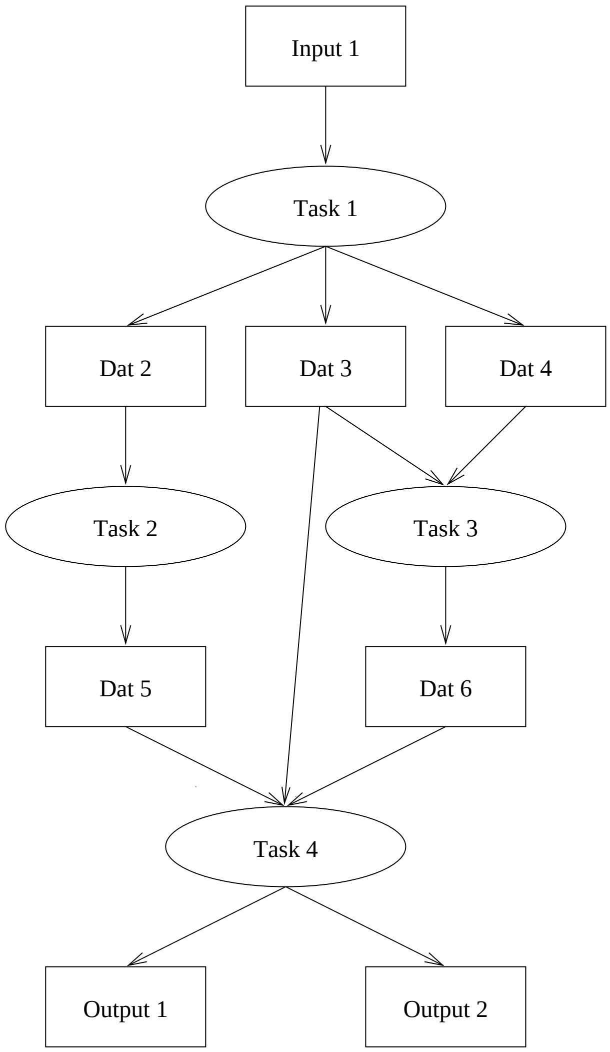


Figura 2.4: Esempio di production

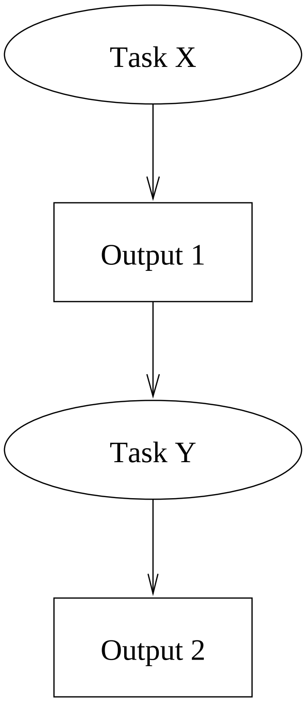


Figura 2.5: Un altro esempio di production

quanto queste operazioni di "setup" dovrebbero essere eseguite tante volte quanti sono gli eventi da elaborare. Per migliorare le performance, gli eventi vengono organizzati in *run*, ognuno costituito da qualche centinaia o migliaia di eventi. In questo modo le operazioni preliminari devono essere eseguite solo per ogni run, e non per ogni evento: questo permette di ridurre notevolmente i tempi totali richiesti per l'elaborazione.

Nel problema considerato si può supporre che la "partizione" dell'insieme di eventi in run sia già stata eseguita: quindi per ogni input di tipo event viene specificata una sequenza di file, ognuno dei quali rappresenta un run.

2.2.2 Architetture usate nell'ambiente HEP

Anche l'ambiente della fisica delle alte energie è una di quella realtà in cui è cambiato il modello di computing: si è verificata la "migrazione" da singoli computer mainframe, a cluster di workstation. Le workstation, infatti, si sono dimostrate adeguate per le elaborazioni tipiche che si devono considerare, con un miglior rapporto prestazioni/prezzo rispetto ai computer di tipo mainframe [7] [20].

Fino a qualche anno fa in quasi tutti gli esperimenti di fisica delle alte energie si è utilizzato un modello standard di workstation, per le attività di simulazione, ricostruzione e analisi: l'architettura Digital-VMS. Si è poi ritenuto più conveniente l'utilizzo di sistemi operativi più "aperti", rispetto al sistema proprietario VMS, in modo da poter utilizzare un numero superiore di architetture hardware. Per questa ragione è stato scelto il sistema operativo Unix, considerando le sue varie implementazioni (Digital-Unix, Hp-Ux, Sun-Solaris, Ibm-Aix, ecc...).

Probabilmente le workstation Unix saranno usate ancora per un certo periodo di tempo, ma i PC, che offrono prestazioni elevate e ad un prezzo molto competitivo, saranno quasi sicuramente la piattaforma di riferimento per il prossimo futuro. È stato provato che sia il sistema operativo Linux, che il sistema operativo Windows NT sono adeguati per eseguire programmi di simulazione/ricostruzione/analisi fisica [10].

2.3 Organizzazione

I fisici che collaborano ad un certo esperimento appartengono a università e istituti di tutto il mondo; il sito principale dell'esperimento e questi altri istituti sono connessi tra loro attraverso una rete geografica (WAN: Wide Area Network): di solito questa non è una rete privata, ma è una rete che deve essere condivisa con altre organizzazioni.

Tutti i dati reali e i dati simulati sono di solito memorizzati su supporti di memoria di massa presenti nel sito principale, disponibili all'intera collaborazione. Vista la grande quantità di eventi da gestire, è possibile memorizzare su dischi solo un sottoinsieme di questi dati, mentre per gli altri si utilizzano supporti di tipo *off-line* (in genere nastri).

Negli altri istituti possono essere presenti sottoinsiemi di questi dati, usati da quei fisici per le loro specifiche analisi. Questi dati vengono copiati su nastri dai dispositivi di storage del sito principale, e vengono poi inviati all'istituto remoto: solo per piccoli volumi si utilizza la Wide Area Network per queste copie.

L'altro possibile approccio per i ricercatori è quello di sottomettere i job di analisi sui calcolatori presenti nel sito principale dell'esperimento, senza quindi la necessità di avere repliche locali dei dati.

Di solito i singoli ricercatori (o gruppi di ricercatori) studiano solo un particolare aspetto del problema fisico generale considerato: a tal fine essi scrivono appositi programmi di analisi, che vengono usati per elaborare specifici sottoinsiemi dei dati disponibili. Tuttavia alcune elaborazioni non vengono "gestite" direttamente dai singoli fisici, ma vengono invece affidate ad una *struttura centralizzata*, che riceve le richieste dagli utenti, "schedula" e sottomette i relativi job sui calcolatori disponibili. Questo approccio è usato in genere per quelle elaborazioni che coinvolgono grandi volumi di dati, per cui è necessaria una grande quantità di risorse di calcolo. In questi casi è infatti fondamentale ottimizzare l'uso delle risorse computazionali, disponibili nel sito principale e/o nei siti remoti. Attraverso una struttura centralizzata di questo tipo viene inoltre ridotto il *manpower* richiesto per la gestione dell'intero sistema [7] [19].

Questa struttura viene usata ad esempio per la produzione di eventi Monte Carlo, ma può essere utilizzata anche per altri tipi di elaborazioni.

2.4 Un esempio: l'esperimento DELPHI

DELPHI (DEtector with Lepton, Photon and Hadron Identification) [3] è uno degli esperimenti LEP.

Il collider LEP, che con una circonferenza di 27 Km è il più grande acceleratore che sia mai stato costruito, è operativo al CERN (Ginevra, Svizzera) dall'agosto 1989. In questo collider gli elettroni e le loro anti-particelle, i positroni, vengono accelerati in direzioni opposte in un condotto in cui è stato creato il vuoto, attorno al quale vi è un anello di magneti. Gli elettroni e i positroni vengono fatti collidere tra loro: viene rilasciata un'energia in un piccolo volume con una densità paragonabile a quella che si presume fosse presente nell'Universo una frazione di secondo dopo il Big Bang. C'è quindi una "sintesi" tra il "molto grande" (l'Universo) e il "molto piccolo" (le particelle elementari).

L'energia iniziale del collider LEP è stata inizialmente posta a circa 91 GeV, in modo che in queste collisioni venissero prodotte particelle Z^0. Le particelle Z^0, e le particelle W^{+-}, entrambe scoperte al CERN nel 1983, sono responsabili delle cosiddette "forze elettrodeboli", responsabili della sopravvivenza del nostro sole. Studiando la creazione e il decadimento delle particelle Z^0, si è studiato e si sta ancora studiando l'attuale teoria delle particelle fondamentali, chiamata *modello standard*. La particella Z^0 ha una vita molto breve: la sua esistenza può essere "dedotta" solo dalla presenza dei frammenti della sua disintegrazione, che possono variare in numero da due fino a un centinaio.

Dalla fine del 1995, è iniziata la seconda fase del collider LEP. L'energia viene aumentata per:

- studiare la produzione delle coppie Z^0Z^0 e delle coppie W^+W^-, per nuovi studi e test del modello standard;

- ricercare nuove particelle, in particolar modo il bosone di Higgs, e/o particelle supersimmetriche.

Si sta procedendo in maniera progressiva:

- nel Novembre 1995 l'energia è stata portata a 130 - 136 GeV;
- nel Luglio - Agosto 1996 l'energia è stata posta a 161 GeV;
- nell'Ottobre - Novembre 1996 a 172 GeV;
- nel 1997 a 183 GeV;
- nel 1998 si sta operando ad un'energia pari a 189 GeV.

Nell'acceleratore LEP i fasci di elettroni e di positroni vengono fatti collidere varie volte. Esistono quattro regioni di intersezione, dove sono posti detector di particelle (ALEPH, DELPHI, L3, OPAL), che misurano le proprietà delle particelle secondarie prodotte nella collisione. Ognuno di questi detector è stato ottimizzato in modo da studiare specifici aspetti e fenomeni fisici.

DELPHI è uno di questi detector. Ha un'alta precisione e un'elevata "granularità", e permette di differenziare le varie particelle secondarie cariche. Attraverso un apposito detector di silicio è possibile individuare anche particelle dalla vita molto breve.

Per la progettazione e la costruzione del detector DELPHI sono stati necessari sette anni.

Attualmente la collaborazione dell'esperimento DELPHI consiste di circa 550 fisici, provenienti da 56 università e istituti di 22 stati.

2.4.1 Elaborazioni in DELPHI

DELSIM (DELphi SIMulation) è il programma standard di simulazione usato nell'esperimento DELPHI. L'unico input di DELSIM è costituito da un *file di titoli* (un input di tipo long term), che permette di selezionare il tipo di eventi da produrre, l'energia, e altri parametri. Si può quindi utilizzare lo stesso programma eseguibile DELSIM per produrre eventi simulati di tutti i tipi: è solo necessario modificare in maniera opportuna il file di titoli. Tuttavia, in alcuni casi particolari (per produzioni speciali) è necessario modificare anche il codice DELSIM, usando quindi un programma non standard. Di solito ogni anno viene rilasciata una nuova versione del programma DELSIM.

DELANA (DELphi ANAlysis) è il programma standard di ricostruzione dell'esperimento DELPHI. DELANA elabora i raw data reali (acquisiti dagli apparati sperimentali) o i raw data simulati (generati dal programma DELSIM), producendo

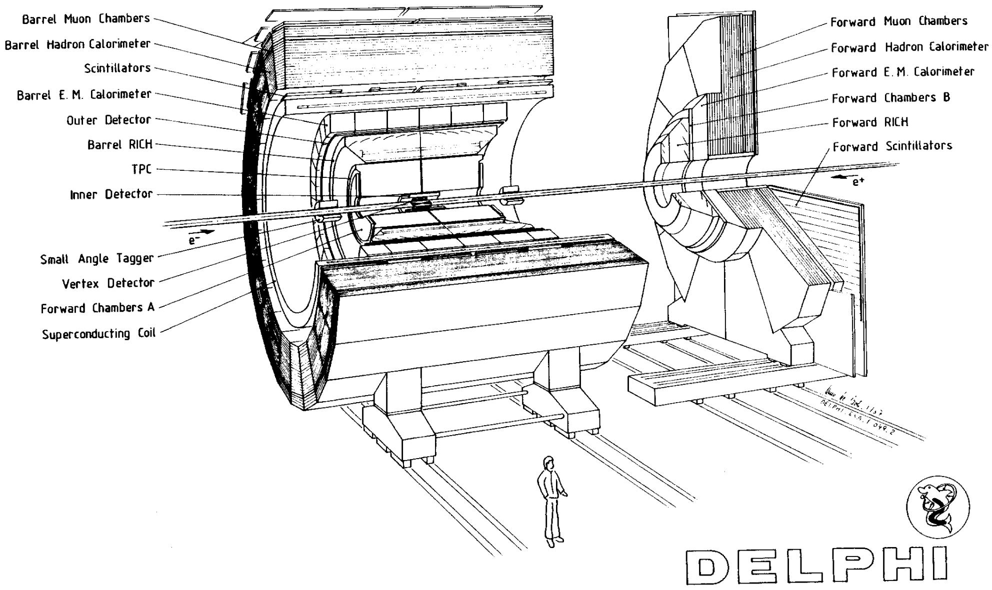


Figura 2.6: Il detector DELPHI

file di tipo *DST* (*Data Summary Tape*). Tutti gli eventi, reali e simulati, di tutti i tipi, vengono analizzati utilizzando lo stesso programma eseguibile DELANA. Di solito ogni anno vengono rilasciate una o più nuove versioni del programma DELANA.

Esistono poi altri programmi standard di analisi, come il programma *SHORT-DST* (l'input di questo programma è costituito da event data di tipo DST, e l'output è costituito da event data di tipo SHORTDST), il programma *MINIDST* (che elabora event data di tipo SHORTDST, producendo event data di tipo MINIDST), ecc...

Si devono considerare poi anche altri programmi "non standard", utilizzati per specifiche analisi. Di solito questi job di analisi sono gestiti direttamente dai singoli fisici, che ne scrivono il codice, e li sottomettono poi sulle varie workstation per la loro esecuzione.

Nell'esperimento DELPHI la produzione degli eventi simulati (usando DELSIM) e la produzione dei corrispondenti file DST (usando DELANA) viene fatta in diversi istituti che fanno parte della collaborazione: Lione, Milano, Padova, Rutherford, Saclay, Stoccolma, ecc... I file DST prodotti in questi istituti vengono memorizzati su nastri, che vengono poi inviati al CERN, il sito principale dell'esperimento.

Al CERN è presente una *farm*, composta da decine di workstation, che viene usata per la ricostruzione e l'analisi dei dati reali. Questa farm viene utilizzata per la produzione di eventi Monte Carlo solamente se non ci sono richieste per l'analisi, o se in quel momento la simulazione è considerata un'attività a più alta priorità.

Nell'esperimento DELPHI non esiste una vera e propria "struttura centralizzata" che gestisce l'esecuzione di tutti questi job: in ogni istituto vi sono una o più persone che si occupano della gestione delle attività di simulazione e ricostruzione, usando le risorse computazionali disponibili localmente. Non esiste però un modello standard a cui fare riferimento: in ogni istituto problemi quali il mapping dei task ai processori, la gestione delle risorse, ecc... vengono gestiti in maniera indipendente e autonoma.

2.5 Esperimenti della prossima generazione

Negli esperimenti HEP della prossima generazione (primi fra tutti gli esperimenti LHC al CERN) si considererà un modello di computing molto diverso rispetto a quello usato negli attuali esperimenti.

Ad esempio, non necessariamente i dati dovranno essere elaborati dove essi sono fisicamente presenti. Infatti un problema che si verifica attualmente è l'*overload* dei calcolatori presenti nel sito principale (dove sono presenti tutti i dati, reali e simulati). Se negli esperimenti della prossima generazione si continuasse ad utilizzare questo approccio, il problema sarebbe ancora più grave, visto che il volume dei dati da elaborare sarà molto più elevato. È più conveniente sfruttare anche le risorse di calcolo presenti nei vari siti, "spostando" i dati se necessario, considerando anche che reti geografiche veloci ed affidabili (come le reti ATM a 622 Mbps) saranno presto disponibili.

Sarà quindi necessario implementare un sistema che, a seconda dei casi, dovrà valutare dove è più conveniente eseguire il job sottomesso dall'utente. Un elemento di questo sistema sarà quindi un *data mover*, che dovrà gestire, in maniera trasparente e dinamica, la copia dei dati tra i vari siti, decidendo le strategie migliori: quando eseguire la copia, che sistema di copia utilizzare (la rete geografica, o la spedizione di nastri attraverso corriere), ecc...

Gli utenti non dovranno più preoccuparsi di conoscere dove i dati sono fisicamente memorizzati: accederanno ad essi usando semplicemente "nomi logici", anziché "nomi fisici" (i pathname dei file).

La "struttura centralizzata" probabilmente non esisterà più: i singoli utenti (o gruppi di utenti autorizzati) potranno definire proprie production, e sottomettere direttamente richieste computazionali al sistema.

Capitolo 3

Il sistema DEMI

Per poter gestire in modo efficiente e funzionale i problemi computazionali tipici del mondo della fisica delle alte energie, ma anche per "trattare" analoghi problemi propri di altri ambienti, viene proposto il modello di un sistema chiamato **DEMI** (Distributed hEterogeneous coMputing envIronment).

Figura 3.1: Il logo del progetto DEMI

3.1 Requisiti

Le caratteristiche e le funzionalità generali richieste da questo sistema si possono riassumere nei seguenti punti:

- DEMI deve gestire un sistema di tipo eterogeneo, costituito da risorse di calcolo di tipo, architettura e prestazioni differenti, distribuite geograficamente tra più siti;
- DEMI deve essere in grado di soddisfare tutte le richieste computazionali sottomesse dagli utenti, che possono essere di tipo e di caratteristiche diverse;
- le performance complessive del sistema devono essere ottimizzate;

- il sistema DEMI deve apparire a tutti gli utenti come un unico grande ambiente di calcolo;
- DEMI deve essere in grado di gestire grandi quantità di dati, che possono essere distribuiti tra più siti diversi. Questi dati devono comunque essere organizzati come un unico grande database logico. Ogni job sottomesso dagli utenti può in teoria accedere all'intero data set;
- il sistema deve essere scalabile, rispetto alle richieste computazionali e rispetto alle risorse di calcolo: DEMI deve essere in grado di gestire facilmente variazioni (anche sensibili) nel volume complessivo dei dati da elaborare, nel numero di job da eseguire, nel numero di calcolatori disponibili, ecc...;
- DEMI deve essere flessibile, in grado di "adattarsi" facilmente ad eventuali variazioni della configurazione del sistema;
- DEMI deve essere *self-managing*, in grado di gestire autonomamente eventuali failure: l'attività di gestione da parte dell'amministratore del sistema deve essere ridotta al minimo indispensabile;
- il sistema deve rimanere nel suo complesso sempre attivo, nonostante guasti o malfunzionamenti hardware e/o software;
- gli utenti del sistema non devono essere costretti a gestire dettagli implementativi non significativi (ad esempio dove i dati sono fisicamente memorizzati);
- il codice dei programmi che vengono eseguiti utilizzando il sistema DEMI deve essere del tutto "normale": non devono essere necessarie particolari istruzioni o particolari configurazioni, e non devono essere imposti vincoli speciali;
- DEMI deve essere un sistema semplice da usare, anche da parte di utenti non particolarmente esperti;
- DEMI non deve essere un sistema specifico per un singolo esperimento di fisica delle alte energie. DEMI, inoltre, deve essere un sistema "general purpose", utilizzabile anche in settori e discipline diverse dall'ambiente HEP, in cui si devono gestire problemi analoghi.

3.2 Ambiente

DEMI deve essere implementato per un ambiente come quello rappresentato in figura 3.2.

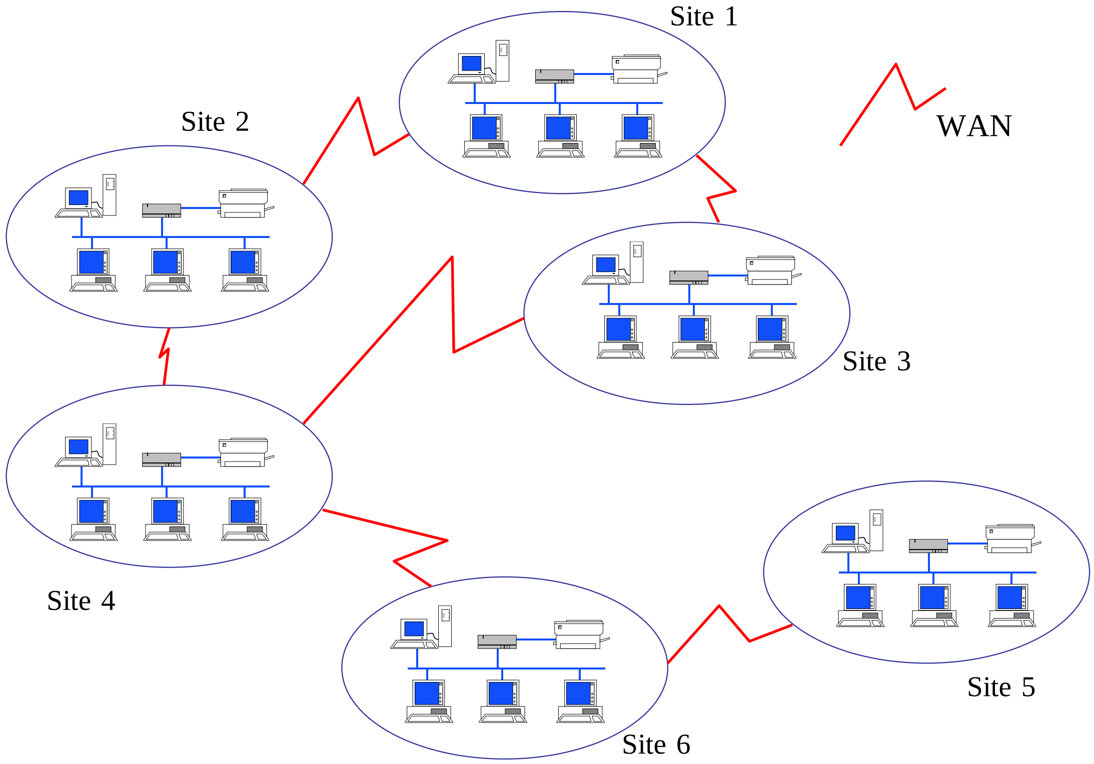


Figura 3.2: Un ambiente tipico, costituito da vari siti connessi attraverso una rete geografica

Deve essere quindi considerato un sistema composto da vari siti, direttamente o indirettamente connessi tra loro attraverso una rete WAN.

In generale i vari siti possono essere molto “diversi” tra loro: possono differire nel numero di computer disponibili, nei modelli di questi sistemi, nell’architettura e nella configurazione della rete locale, ecc.. Ogni sito costituisce un *metasistema*: un sistema di elaborazione distribuito, costituito da sistemi indipendenti ed eterogenei, collegati tra loro attraverso una rete locale, che può essere implementata in diversi modi (Ethernet, Fast Ethernet, FDDI, ATM, Gigabit Ethernet, ecc...).

Si può assumere che la comunicazione tra i vari sistemi, posti nello stesso sito o in siti diversi, è sempre implementata attraverso i protocolli TCP/IP; non è invece richiesto il supporto per altri protocolli (DECnet, SNA, ecc...).

3.2.1 Sistemi supportati

Il sistema DEMI deve essere in grado di gestire:

- Workstation Unix (il sottoinsieme più grande possibile);
- PC e workstation con sistema operativo Windows NT.

Deve comunque essere possibile effettuare facilmente il porting dell’applicazione ad eventuali nuove architetture, con un elevato rapporto prestazioni/prezzo, che si rendessero disponibili. Infatti è fondamentale poter disporre della massima potenza computazionale, utilizzando le risorse economiche disponibili.

Solitamente ogni computer deve essere considerato come un calcolatore “general purpose”, di cui il sistema DEMI può utilizzare solamente i cicli macchina liberi. Quindi i vari calcolatori devono ritenersi disponibili al sistema DEMI solamente quando essi non sono utilizzati dai rispettivi utenti. Si rende quindi necessaria una gestione delle priorità: l’utente di una workstation deve aver sempre la precedenza nell’utilizzo di tale risorsa, rispetto al sistema DEMI. Quando un utente vuole utilizzare la “propria” workstation, gli eventuali job DEMI in esecuzione su questo computer devono essere sospesi, ed eventualmente “migrati” su un altro calcolatore. Questa procedura deve essere effettuata in maniera automatica e trasparente: l’utente non deve essere obbligato ad utilizzare un comando esplicito per “reclamare” l’uso della sua workstation, ma deve essere il sistema a verificare automaticamente la “presenza” dell’utente (ad esempio rilevando l’attività sulla tastiera, controllando quali sono i processi in esecuzione, ecc...).

3.3 Metodologia di sviluppo

Considerando una qualsiasi applicazione software che deve essere sviluppata, i requisiti e le caratteristiche che in genere sono richiesti sono molteplici [28]: affidabilità, robustezza (il software deve comportarsi in maniera adeguata anche in presenza di situazioni anomale), usabilità (l'utente deve essere in grado di utilizzare l'applicazione senza particolari difficoltà), verificabilità (è necessario essere in grado di dimostrare che il sistema realizzato effettivamente risponde alle caratteristiche e alle funzionalità richieste), manutenibilità, riusabilità, interoperabilità (l'applicazione sviluppata deve poter facilmente integrarsi con altre applicazioni, che realizzano particolari funzionalità), ecc...
È chiaro, quindi, che per soddisfare tutti questi requisiti è necessario utilizzare una precisa metodologia nello sviluppo del software, soprattutto nel caso in cui il sistema che deve essere implementato sia particolarmente complesso.

Nella realizzazione del sistema DEMI si è utilizzata la metodologia chiamata *ciclo di vita a cascata* [28]. Questo modello prevede che lo sviluppo dell'applicazione si articoli in una serie di fasi, poste in cascata: ogni fase riceve ingressi dalla fase precedente, e produce risultati che vengono poi usati nella fase successiva. Una volta che sono stati ottenuti i risultati di una fase, e questi sono stati convalidati, si passa alla fase successiva. Il numero e i tipi di fasi in cui si articola lo sviluppo di un sistema non sono definiti a priori, in quanto dipendono da vari fattori: le caratteristiche e la complessità del sistema da realizzare, il ruolo del "produttore" dell'applicazione (può coincidere oppure no con il committente), l'utilizzo che si dovrà fare dell'applicazione (uso esclusivo da parte del committente, o prodotto che potrà essere utilizzato anche da altri utenti), ecc...
Nello sviluppo del sistema DEMI si sono considerate le seguenti fasi iniziali:

- analisi preliminare del problema;
- analisi e specifica dei requisiti;
- progettazione;
- implementazione.

3.3.1 Analisi preliminare del problema

Per definire in maniera chiara e non ambigua il dominio del problema, si è provveduto innanzittutto alla stesura di un **glossario**, in cui vengono definite le "parole fondamentali". Durante lo sviluppo del sistema, tale glossario è stato continuamente aggiornato con i nuovi termini introdotti.

Il glossario è riportato in appendice A.

Si sono poi analizzati alcuni sistemi e applicazioni già esistenti, che realizzano (almeno parzialmente) alcune funzionalità richieste nel sistema considerato. Queste applicazioni sono descritte e analizzate nel capitolo 5 e in appendice B.

3.3.2 Analisi e specifica dei requisiti

Si sono poi definiti in dettaglio le funzionalità e i requisiti dell'applicazione da realizzare, definendo "cosa" deve essere fornito dal sistema.

Esistono diversi modelli per la specifica formale dei requisiti [28]: il modello entità-relazioni, i diagrammi di flusso dei dati, gli automi a stati finiti, le reti di Petri, la logica temporale del primo ordine, ecc... Queste specifiche formali consentono di definire le funzionalità richieste senza ambiguità, e inoltre possono essre "manipolate" automaticamente per valutarne la completezza, per analizzare le relazioni che esistono tra i vari concetti definiti, ecc... Tuttavia per il sistema DEMI si è preferito considerare una specifica di tipo informale, scritta in linguaggio naturale, sostanzialmente per garantire la leggibilità anche da parte di persone che non hanno specifiche competenze in materia.

Il risultato della fase di analisi e specifica dei requisiti del sistema DEMI, è il cosiddetto documento URD (*User Requirement Document*), presentato in [1]. I requisiti del sistema DEMI sono comunque descritti nel capitolo 4.

3.3.3 Progettazione

All'analisi e specifica dei requisiti è seguita la fase di progettazione, cioè le attività che consentono di realizzare effettivamente l'applicazione. Oltre a definire l'architettura software del sistema, si è data particolare enfasi al problema della gestione dei dati.

La progettazione del sistema DEMI è presentata nei capitoli 6 e 7 .

3.3.4 Implementazione

Alla progettazione è seguita la realizzazione di un primo prototipo, e la conseguente verifica della sua funzionalità.

Questa fase dello sviluppo del sistema DEMI è descritta nel capitolo 8.

3.4 Gestione della documentazione

I vari documenti che sono stati prodotti durante lo sviluppo del sistema DEMI sono stati organizzati in un server web [1], in cui sono state inserite anche altre informazioni: discussioni relative alla realizzazione del progetto, link ad altri progetti correlati, bibliografia, ecc... Nelle figure 3.3 e 3.4 sono presentate alcune di queste pagine ipertestuali.

DEMI

[Home | Persons | Schedule | Archive | Search | Discussions | Miscellaneous]

HOME

! Welcome to the home page for project DEMI !

DEMI is the proposal for a new system capable to manage computations in a distributed and heterogeneous computing environment.
It has been planned to satisfy the specific requirements of the HEP (High Energy Physics) environment, but it can be applicable to other disciplines as well.

Take a look at What's New in our web.

What's New

The following is a list of recent additions to our web. Whenever we publish a paper, write a specification, submit a status report, or add anything else to our web, we'll put a notice here. The most recent changes are listed first, and each item is linked to the page with the updated content.

May 29, 1998

- New document: "Objectivity Tecnical Overview"

May 20, 1998

- Testing the first prototype of the DEMI system

May 11, 1998

- Implementing a first prototype of the DEMI system

April 10, 1998

- New document "Specifica dell'interfaccia delle classi"

March 3, 1998

- New document "Architettura del sistema"

February 10, 1998

Figura 3.3: Una pagina del server WWW

DEMI

[Home | Persons | Schedule | Archive | Search | Discussions | Miscellaneous]

ARCHIVE

This page contains links to documents authored by project members, to software programs and components developed by the project, and to other tools we've found useful and collected here for your convenience.

Each entry has a title, a summary, a list of authors, a date of publication or when it was last updated, and links to the material in various formats.

Documents

Introduction to the problem

The initial document that describes the most important features and characteristics required for the DEMI system.

```
By:           Massimo Sgaravatto
Last updated: December 3, 1997
Formats:      .ps (682 k), .gz (138 k)
```

Glossary

Up-to-date glossary that describes the "domain" of the problem. This document is always kept in sync with updates of the other documents.

```
By:           Massimo Sgaravatto
Last updated: November 10, 1997
Formats:      HTML
```

User Requirement Document

The document where all the requirements are specified.

```
By:           Massimo Sgaravatto
Last updated: December 3, 1997
Formats:      .ps (536 K), .gz (111 K)
```

Ambienti visuali per la programmazione parallela

Analisi di alcuni ambienti visuali per la programmazione parallela

```
By:           Massimo Sgaravatto
Last updated: November 24, 1997
Formats:      .ps (44 M)
```

Figura 3.4: Un'altra pagina del server WWW

Capitolo 4

Funzionalità del sistema DEMI

In questo capitolo vengono definite le caratteristiche e le funzionalità che devono essere implementate nel sistema DEMI.

Poiché il sistema non deve essere specifico all'ambiente HEP, in questa descrizione si utilizza una terminologia "generale", non propria dell'ambiente della fisica delle altre energie.

Ulteriori informazioni riguardo l'analisi e la specifica dei requisiti si possono trovare nello User Requirement Document [1].

4.1 Introduzione

I vari siti possono differire tra loro sotto diversi aspetti: nel tipo e nel numero di risorse computazionali, nel numero di utenti, ecc... In particolare nei vari siti si possono voler considerare diverse "politiche" di gestione delle risorse locali. Ad esempio, in certe realtà le workstation potrebbero essere dedicate alle elaborazioni DEMI, mentre in altri ambienti i calcolatori potrebbero essere utilizzati dal sistema DEMI solo se non utilizzati dagli altri utenti.

Per questi motivi è opportuno separare l'amministrazione del sistema in due livelli. In ogni sito la gestione è affidata ad un *local administrator*: suo compito è quello di gestire e controllare le attività locali del sistema DEMI (in particolare la gestione delle risorse). L'amministrazione del sistema globale è invece affidata ad un *global administrator*, responsabile nel gestire e coordinare l'intero sistema: il global administrator deve occuparsi di attività quali la configurazione dell'ambiente, la gestione dei database globali, ecc...

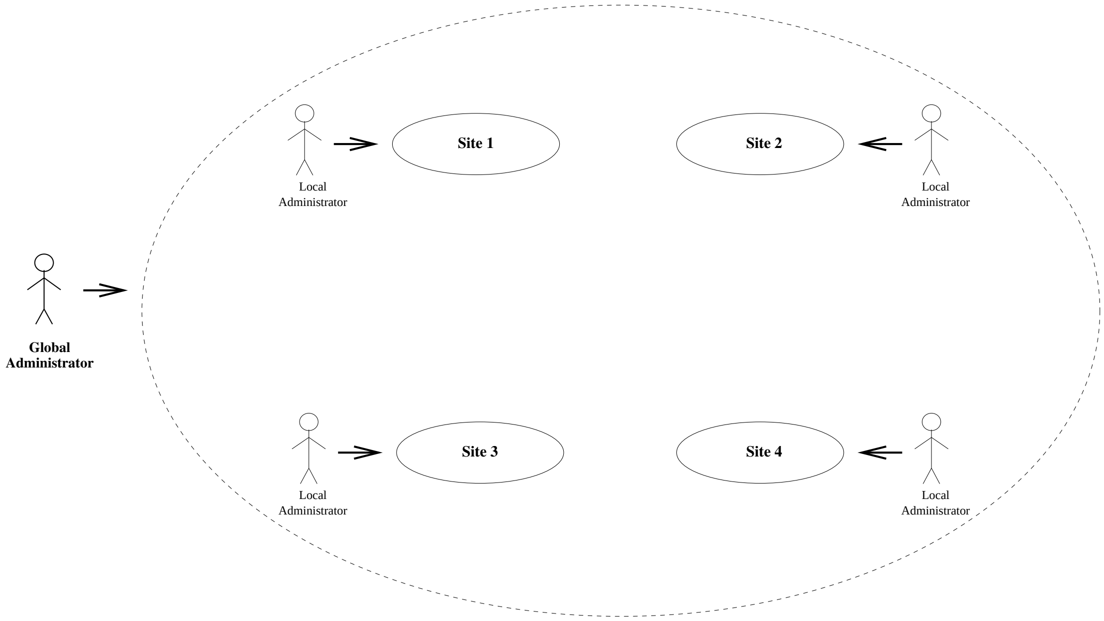


Figura 4.1: Global administrator e local administrator

4.2 Definizione dell'ambiente globale

Per definire l'ambiente globale in cui il sistema DEMI deve operare, è necessario specificare:

1. la configurazione del sistema;
2. le architetture dei calcolatori utilizzati;
3. i modelli di tape device usati.

4.2.1 Gestione della configurazione del sistema

Il global administrator è responsabile nella definizione della configurazione del sistema DEMI: devono essere specificati i siti componenti il sistema, e le loro caratteristiche.
Per ogni sito è necessario definire:

- il nome del sito (es. *Padova*);
- una descrizione (es. *Sito dell'INFN sezione di Padova*);
- un "indirizzo" che permetta di accedere al sito considerato.

Il global administrator deve essere in grado di:

- aggiungere al sistema nuovi siti;
- rimuovere siti dal sistema DEMI;
- modificare informazioni relative a siti precedentemente inseriti.

4.2.2 Gestione delle architetture dei computer

Come detto, i computer impiegati nel sistema DEMI possono essere di tipo e caratteristiche diverse. Questi calcolatori possono essere classificati a seconda della loro *architettura*: il tipo di architettura definisce la compatibilità binaria per i file eseguibili. Quindi due computer hanno la stessa architettura se e solo se per ogni task è possibile usare nei due sistemi gli stessi file eseguibili per l'esecuzione. Solitamente l'architettura è determinata dal modello del computer e dal sistema operativo considerato.

Le informazioni relative alle architetture dei calcolatori devono essere memorizzate in maniera persistente. A tal fine si può considerare un'apposita struttura dati, chiamata **Architecture Database**, in cui ogni elemento è definito dai seguenti campi:

- un identificatore (es. *alpha-win32*);
- una descrizione (es. *Computer con processore alpha, con sistema operativo Windows NT*);
- altre eventuali informazioni.

Il global administrator deve poter:

- definire nuove architetture di calcolatori;
- rimuovere elementi dall'Architecture Database;
- modificare informazioni relative ad architetture prima definite;
- visualizzare gli elementi definiti nell'Architecture Database.

4.2.3 Gestione dei modelli di tape device

I dischi in genere non sono sufficienti per la memorizzazione di tutti i dati. Si possono allora usare nastri, che hanno un'elevata capacità di memorizzazione, e costi relativamente contenuti. I nastri, e i relativi tape device (i dispositivi che permettono di scrivere/leggere dati sui/dai nastri) possono essere di tipo diverso (DLT, DAT, Exabyte 8mm, ecc...).

Il global administrator deve gestire attraverso un apposito database, chiamato **Tape Device Model Database**, i modelli di tape device (e dei relativi nastri) utilizzati nel sistema DEMI.
Ogni elemento di questo database viene definito dalle seguenti informazioni:

- un identificatore (es. *DLT*);
- una descrizione (es. *tape device DLT, per nastri TZ87*);
- la capacità di un nastro di questo modello, ovvero la massima quantità di dati che è possibile scrivere in un nastro di questo tipo (es. *30 GB*);
- altre eventuali informazioni.

Le operazioni che devono poter essere effettuate dal global administrator sono:

- la definizione di nuovi modelli di tape device;
- la cancellazione di elementi dal Tape Device Model Database;
- la modifica di informazioni relative ad elementi definiti nel database;
- la visualizzazione degli elementi definiti nel Tape Device Model Database.

Come rappresentato in figura 4.2, l'Architecture Database e il Tape Device Model Database sono due database globali, di cui esiste un'unica istanza per tutto il sistema DEMI. Questi database possono essere modificati solamente dal global administrator, mentre gli utenti dei vari siti (local administrator compresi) vi possono accedere solo per operazioni di lettura.

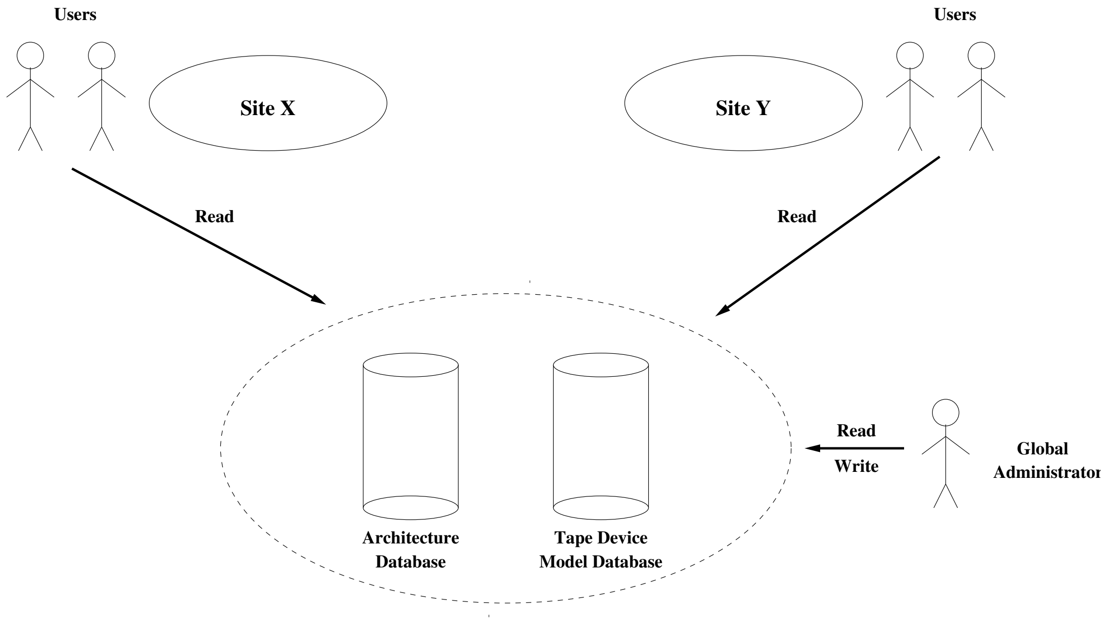


Figura 4.2: L'Architecture Database e il Tape Device Model Database

4.2.4 Global administrator

Possono esistere più global administrator, responsabili della gestione del sistema DEMI a livello globale. Le informazioni relative a questi amministratori devono essere memorizzate in un'apposita struttura dati, chiamata **Global Administrator Database**. Ogni elemento di questo database è definito dai seguenti campi:

- un identificatore, chiamato **user identifier** (es. *sgaravat*);
- la password (criptata) che l'administrator deve specificare quando vuole gestire il sistema DEMI;
- l'indirizzo e-mail dell'amministratore (es. *sgaravat@domain*);
- altre eventuali informazioni.

Su questo database possono essere fatte, da parte di un global administrator, le classiche operazioni di inserimento, cancellazione, modifica, ricerca e visualizzazione.

4.3 Definizione dell'ambiente locale

I local administrator sono responsabili dell'amministrazione dei corrispondenti siti, in particolare della gestione delle risorse locali.
Si devono considerare tre tipi di risorse:

- i computer, utilizzati per le elaborazioni sottomesse dagli utenti;
- i disk buffer, usati per la memorizzazione dei dati;
- i tape device, utilizzati per la lettura/scrittura dei dati da/su nastri.

In ogni sito è necessario memorizzare i dati relativi a queste tre classi di risorse, considerando tre specifiche strutture dati:

- il **Local Computer Database**;
- il **Local Buffer Database**;
- il **Local Tape Device Database**.

4.3.1 Gestione dei computer

Ogni elemento del Local Computer Database definisce un calcolatore presente nel sito, utilizzato dal sistema DEMI, e viene rappresentato da vari campi, quali:

- il nome o l'indirizzo IP del computer (es. *ntde01.pd.infn.it*);
- la descrizione (es. *alpha station 3000/600, con sistema operativo Windows NT Server 4.0*);
- l'architettura di questo computer (es. *alpha-win32*): questa architettura deve essere definita nell'Architecture Database;
- un valore numerico, chiamato 'performances', che definisce le prestazioni di questo computer: più alto è il valore di questo parametro, migliori sono le prestazioni. Il valore di questo campo può essere ottenuto attraverso un opportuno benchmark, e può quindi essere espresso, a seconda dei casi, in MIPS, MFLOPS, SPECINT, SPECFP, ecc... Nell'ambiente HEP si considera in genere il *CERN benchmark* [4], in cui si valutano le performance dei calcolatori considerando una serie di elaborazioni specifiche di fisica delle alte energie (l'unità di misura in questo caso è la *CERN Unit*).

Il local administrator deve essere in grado di definire nuovi computer nel Local Computer Database, di rimuovere calcolatori, di "disabilitare" computer (ovvero eliminare questi calcolatori dalla configurazione del sistema, senza però eliminarli dal database), di "riabilitare" computer, di modificare informazioni di un elemento presente nel Local Computer Database, di verificare lo stato dei vari calcolatori presenti nel sito, ecc...

4.3.2 Gestione dei disk buffer

I disk buffer vengono utilizzati dal sistema DEMI per la memorizzazione dei file di dati definiti nel Local Data Database (vedi par. 4.4.2), e dei file "temporanei" prodotti nei vari processing (dati che vengono cancellati quando non sono più necessari). Un disk buffer può essere un "intero" disco, o solamente la parte di un disco.

Si può supporre che esista un file system comune (implementato attraverso sistemi quali NFS [50], AFS [51], DCE/DFS [38], Samba [31], ecc...), per cui ogni computer del sito può accedere direttamente ad ogni buffer.
Ogni elemento del Local Buffer Database rappresenta uno di questi disk buffer, ed è descritto dai seguenti campi:

- un identificatore (es. *disk1*);
- il computer dove il disk buffer è fisicamente installato (es. *osde01.pd.infn.it*): questo elemento deve essere definito nel Local Computer Database;
- il pathname con cui si "accede" al buffer (es. */DEMI/disk1*);
- lo spazio totale disponibile in questo disk buffer per la memorizzazione di file DEMI (es. 2 Gbyte);
- altre eventuali informazioni.

Il local administrator deve essere in grado di aggiungere nuovi disk buffer al sistema DEMI, di rimuovere buffer dal sistema, di modificare informazioni di elementi presenti nel Local Buffer Database, di verificare lo stato dei vari buffer presenti nel sito (spazio usato, spazio disponibile, ecc...), ecc...

4.3.3 Gestione dei tape device

Il Local Tape Device Database è usato per registrare le informazioni relative ai tape device presenti nel sito, utilizzati dal sistema DEMI per la lettura/scrittura di file di dati da/su nastro. Ogni elemento di questo database è definito da varie informazioni quali:

- un identificatore (es. *tapedev1*);
- il corrispondente modello di tape device (es. *DLT*): questo modello deve essere definito nel Tape Device Model Database;
- il computer dove il device è fisicamente installato (es. *osde01.pd.infn.it*);
- il pathname con cui si "accede" al tape device (es. */dev/rmt0h*);
- altre eventuali informazioni.

Il local administrator deve poter essere in grado di aggiungere nuovi tape device al sistema DEMI, di rimuovere tape device dal sistema, di modificare informazioni di un elemento presente nel Local Tape Device Database, di rilevare lo stato dei vari tape device presenti nel sito, ecc...

Come descritto in figura 4.3, in ogni sito esistono specifiche istanze locali del Local Computer Database, del Local Buffer Database e del Local Tape Device Database. I local administrator possono accedere a questi database con operazioni di lettura-scrittura, mentre gli altri utenti del sito possono effettuare solo operazioni di lettura. I Local Computer Database e i Local Tape Device sono "in relazione" con i database globali Architecture Database e Tape Device Model Database, rispettivamente per le definizioni delle architetture dei computer e dei modelli di tape device.

4.3.4 Gestione degli utenti

In ogni sito del sistema DEMI deve essere utilizzato uno specifico database, chiamato **User Database**, per memorizzare le informazioni relative agli utenti DEMI del sito locale. Ogni utente deve essere definito nel database attraverso i seguenti campi:

- uno **user identifier** (es. *rossi*);
- la password (criptata) che l'utente deve specificare quando vuole "accedere" alle funzionalità offerte dal sistema DEMI;
- l'indirizzo e-mail dell'utente (es. *rossi@localdomain*);
- il "ruolo" di questo utente:

  la persona considerata può essere un utente generico, un local administrator, un operatore (responsabile del mount dei nastri negli appositi tape device). Un utente può avere anche più ruoli (ad esempio può essere allo stesso tempo un utente generico e un operatore);

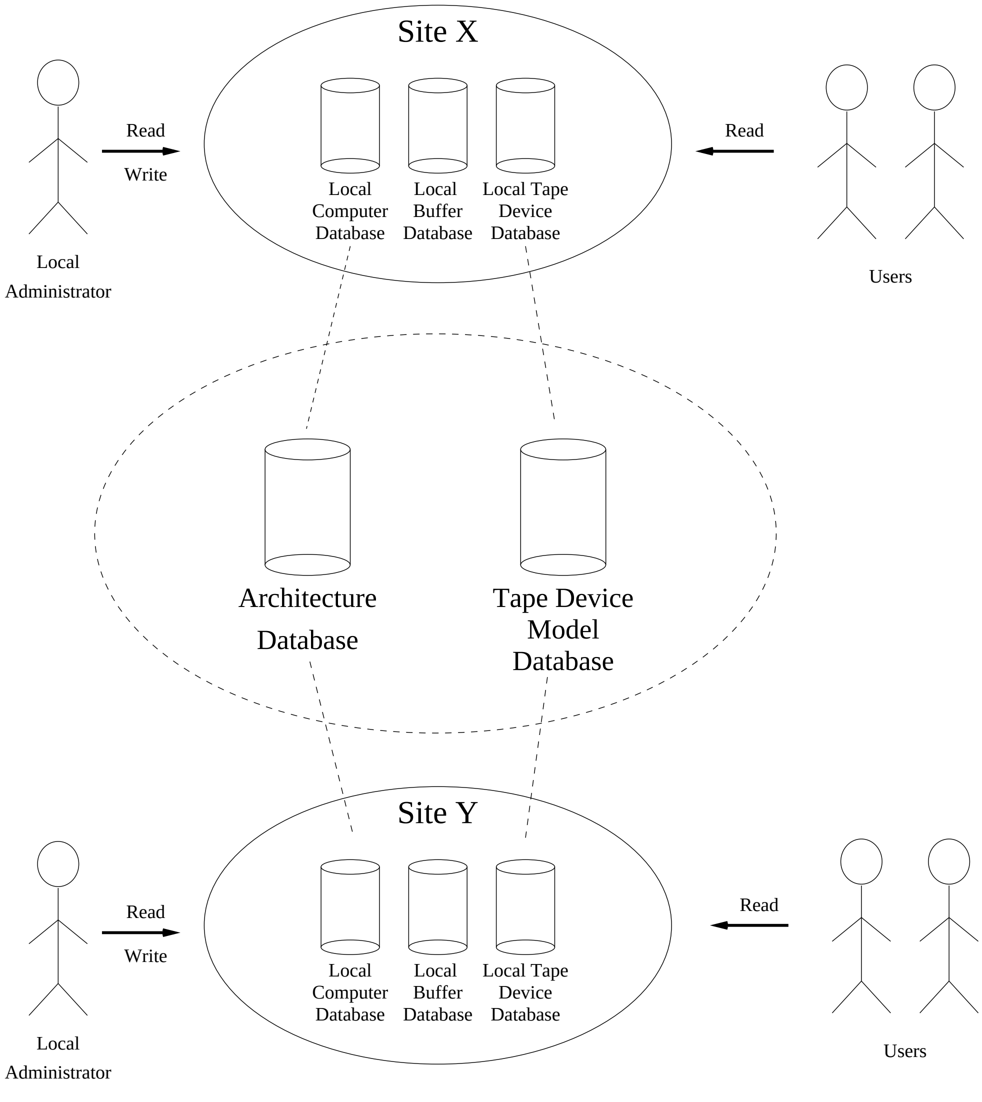


Figura 4.3: I database usati per la gestione delle risorse presenti nei vari siti

- altre eventuali informazioni.

Il local administrator può effettuare sullo User Database operazioni di inserimento, cancellazione, modifica, ricerca e visualizzazione.

4.4 Gestione dei dati

In generale, in ogni sito del sistema DEMI possono essere presenti file di dati, memorizzati su disk buffer (vedi par. 4.3.2) e/o su nastri (vedi par. 4.3.3).

Anche se i dati possono essere distribuiti geograficamente tra i vari siti, essi devono essere organizzati in un unico database logico, chiamato **Data Database**.

Per "accedere" ad un file di dati, l'utente non deve preoccuparsi di dove esso è fisicamente memorizzato, e non deve nemmeno conoscere il relativo pathname o identificatore del nastro: è necessario specificare semplicemente un "nome logico", chiamato **file identifier**, che permette di identificare senza ambiguità il file richiesto.

La struttura del sistema di gestione dei dati è rappresentata in figura 4.4: il Data Database consiste di un **Global Data Database** "centrale" e, in ogni sito del sistema DEMI, di un **Local Data Database**.

4.4.1 Global Data Database

Nel Global Data Database sono memorizzate le informazioni relative a tutti i file di dati gestiti dal sistema DEMI. Ognuno di essi viene descritto dai seguenti campi:

- il corrispondente file identifier;
- la descrizione del tipo di dati memorizzati nel file (es. *QQPS, 183 GeV, 97_1 MC*);
- il nome dei siti in cui il file è fisicamente presente (uno stesso file può essere presente in più siti);
- altre eventuali informazioni.

4.4.2 Local Data Database

In ogni sito del sistema DEMI è presente un Local Data Database, in cui vengono memorizzate informazioni relative ai file di dati fisicamente presenti nel sito locale. Ognuno di questi file è rappresentato nel database dai seguenti campi:

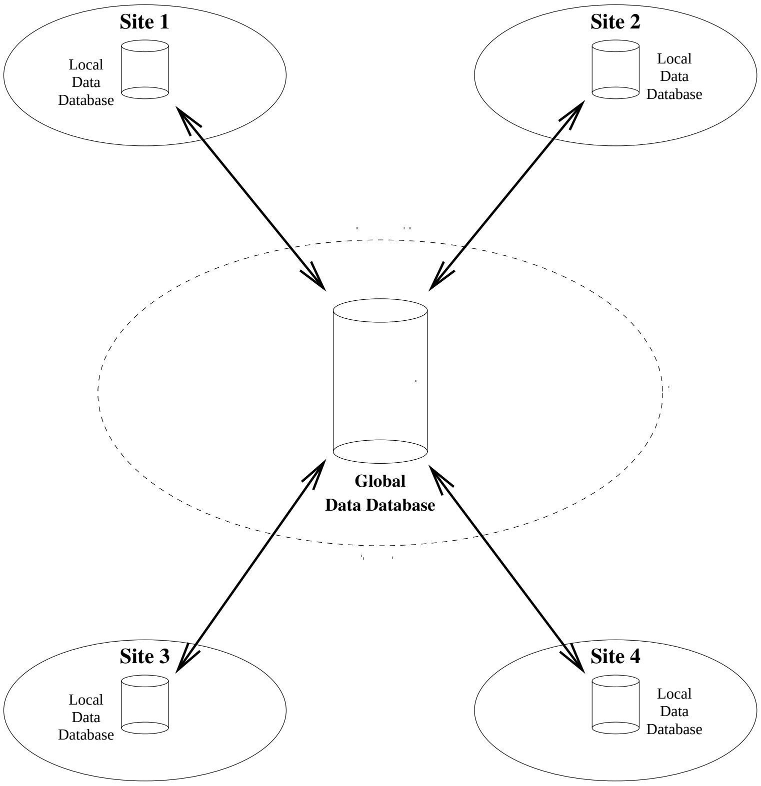


Figura 4.4: Data Database

- il file identifier;
- per ogni "occorrenza" di questo file nel sito (il file può essere presente in più dischi e/o nastri), sono specificate le seguenti informazioni:
  - il tipo di supporto di memoria di massa utilizzato per la memorizzazione del file, che può essere:
    * un disk buffer;
      in questo caso è necessario specificare l'identificatore di questo buffer (vedi par. 4.3.2);
    * un nastro;
      in questo caso è necessario indicare l'identificatore di questo nastro (vedi par. 4.4.3);
- altre eventuali informazioni.

Si supponga, ad esempio, che un certo file di dati sia presente nel sito *SiteX*, memorizzato su un disk buffer. Nel Global Data Database deve essere presente un elemento per questo file, che indica appunto che il file è presente nel sito *SiteX*. Nel Local Data Database del sito *SiteX*, deve essere definito un elemento per questo file, con l'indicazione che esso è memorizzato su un disk buffer, di cui viene dato l'identificatore.

Quindi nel Global Data Database vi è un elemento per ogni file di dati gestito dal sistema DEMI, mentre nel Local Data Database sono memorizzate informazioni solamente per quei file che sono presenti nel corrispondente sito.

Ogni utente del sistema DEMI deve poter interrogare il Data Database, utilizzando in queste query uno o più degli attributi descritti.

I vari campi del Data Database vengono aggiornati automaticamente dal sistema dopo ogni processing, per registrare i file di output prodotti: non devono essere gli utenti ad aggiornare manualmente il database. Ogni administrator deve comunque poter essere in grado di gestire direttamente il database, per inserire nuovi elementi, per modificare attributi di elementi presenti nel database, per cancellare elementi, ecc...

4.4.3 Gestione dei nastri

In ogni sito le informazioni relative ai nastri in cui vengono memorizzati i file di dati definiti nel Data Database, devono essere gestite attraverso un'opportuna struttura dati, chiamata **Local Tape Database**. In questo database ogni nastro è rappresentato dalle seguenti informazioni:

- un identificatore, chiamato **tape identifier** (es. *tape1*);
- il modello del nastro (che deve essere definito nel Tape Device Model Database);
- la descrizione del tipo dei file memorizzati nel nastro. Infatti, per rendere più agevoli le operazioni di accesso ai dati, si pone il vincolo che i file che si memorizzano in un nastro siano tutti dello stesso tipo;
- altre eventuali informazioni.

4.5 Definizione delle elaborazioni

La definizione delle elaborazioni che dovranno essere eseguite dal sistema si articola in tre fasi:

1. definizione di tipi di task;
2. definizione di production, utilizzando tipi di task precedentemente definiti;
3. submit di richieste computazionali (utilizzando production precedentemente definite), per il processing di dati definiti nel Data Database.

4.5.1 Definizione di tipi di task

Gli utenti del sistema DEMI devono essere in grado di definire tipi di task. Un task rappresenta un'attività elementare di elaborazione, che gestisce un particolare aspetto del problema complessivo. Un task può avere zero o più input, e zero o più output.

Deve essere messo a disposizione degli utenti un apposito tool, per definire i tipi di task attraverso rappresentazioni grafiche, come quella mostrata in figura 4.5.

Nella rappresentazione grafica l'ellisse rappresenta il tipo di task, mentre i rettangoli rappresentano i corrispondenti input/output (nell'esempio di figura il tipo di task *TaskTypeX* ha due input: *Input1* e *Input2*, e tre output: *Output1*, *Output2* e *Output3*).

Le definizioni dei tipi di task devono essere memorizzate in uno specifico database, chiamato **Task Database**. In questo database ogni tipo di task viene rappresentato dai seguenti campi:

- il nome del tipo di task (es. *TaskTypeX*);

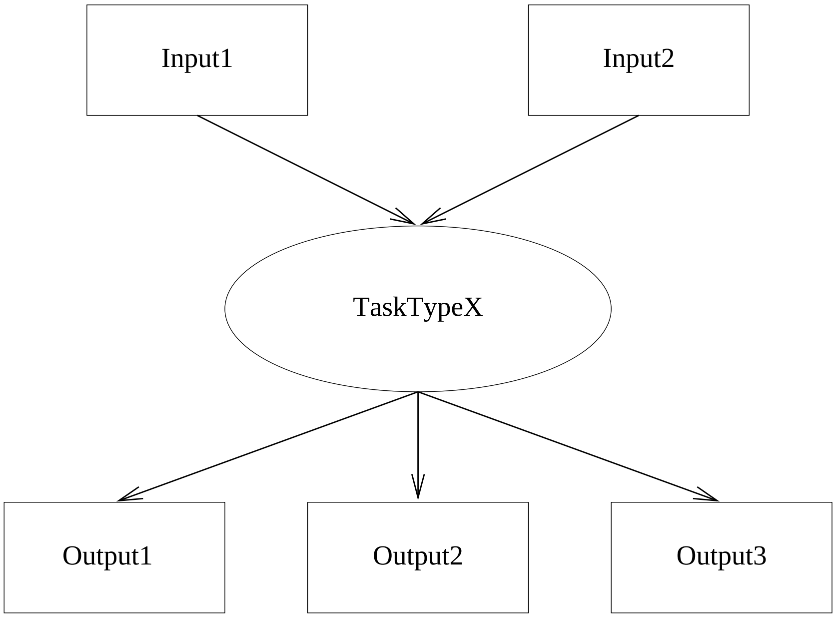


Figura 4.5: Esempio di rappresentazione grafica di un tipo di task

- la descrizione (es. *Task che gestisce l'attività X*);
- lo user identifier dell'utente che ha definito il tipo di task;
- gli input del tipo di task.

  È necessario specificare:

  - il numero di questi input;
  - il loro nome (es. *Input1*, *Input2*, ecc..);

- gli output del tipo di task.

  È necessario specificare:

  - il numero di questi output;
  - il loro nome (es. *Output1*, *Output2*, ecc..);

- la lista delle architetture di calcolatori per le quali il tipo di task considerato è stato implementato (le architetture da considerare sono quelle definite nell'Architecture Database).

  Per ogni elemento di questa lista è necessario definire:

  - i file necessari per l'esecuzione del task (programma eseguibile, script che definiscono variabili logiche, ecc...); non devono invece essere specificati i file di input e di output;
  - la linea di comando che deve essere specificata per eseguire il task.

    Nel programma utilizzato per l'esecuzione del task, si fa riferimento agli input e agli output attraverso dei "parametri formali" (ad esempio con il primo parametro si può fare riferimento all'input *Input1*, con il secondo parametro all'input *Input2*, ecc...). Quando si deve eseguire il task, utilizzando file di input e producendo file di output, i nomi di questi file di input/output costituiscono i "parametri attuali", che devono essere associati ai corrispondenti "parametri formali". I parametri attuali vengono specificati nella linea di comando usata per eseguire il task.

    Ad esempio, per il task di figura 4.5, nella linea di comando dovranno essere specificati almeno cinque parametri (è possibile che ne siano necessari anche altri): due per i due input *Input1* e *Input2*, e gli altri tre per gli output *Output1*, *Output2* e *Output3*. Ad esempio, se per l'input *Input1* deve essere usato il file *file1*, per l'input *Input2* si deve utilizzare il file *file2*, e per i tre output *Output1*, *Output2* e *Output3* si devono produrre rispettivamente i file con nome *out1file*, *out2file* e *out3file*, la linea di comando sarà del tipo:

    ```
    TaskTypeXExecFile    file1 file2 out1file \
                         out2file out3file
    ```

    È quindi necessario definire la corrispondenza tra i numeri ordinali dei parametri, e i corrispondenti nomi degli input-output;
  - il parametro 'cost of the task';

    questo valore numerico rappresenta la stima del tempo necessario per l'esecuzione del task sull'architettura considerata. Non si tratta di un vero e proprio valore di tempo (ad esempio un valore pari a '10' non rappresenta 10 secondi, o 10 minuti): questo parametro è utilizzato per confrontare le diverse performance delle varie architetture nell'esecuzione del task considerato (un'architettura può essere più adatta rispetto ad un'altra nell'eseguire un certo task).

    Ovviamente possono esistere più calcolatori della stessa architettura con diverse prestazioni: questo aspetto viene considerato nel problema dello scheduling (vedi par. 4.7);

- altre eventuali informazioni.

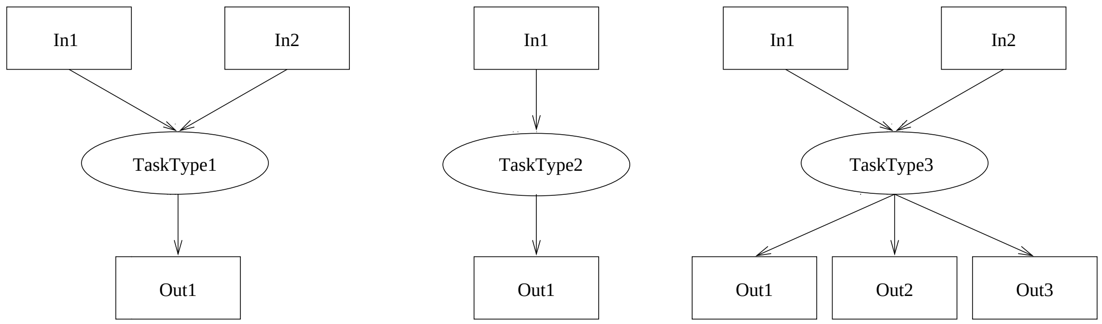


Figura 4.6: Esempi di tipi di task

Gli utenti del sistema DEMI devono poter essere in grado di definire nuovi tipi di task, di cancellare dal Task Database elementi precedentemente definiti, di modificare informazioni relative a tipi di task definiti nel database, di visualizzare gli elementi memorizzati nel Task Database, ecc...

4.5.2 Definizione di production

Utilizzando i tipi di task definiti nel Task Database, gli utenti del sistema DEMI possono definire production (produzioni): una production rappresenta un'elaborazione "complessa", composta da una serie di task in relazione tra loro. Quindi se un task gestisce un particolare aspetto del problema considerato, in una production si considera tale problema nella sua complessità.

Una production può essere composta da uno o più task, che possono anche essere (tutti o in parte) dello stesso tipo.

Anche in questo caso deve essere messo a disposizione degli utenti un apposito tool grafico per la definizione di production.

In figura 4.6 sono rappresentati tre tipi di task (*TaskType1*, *TaskType2* e *TaskType3*), che vengono utilizzati per la definizione della production di figura 4.7: il task *Task1* è di tipo *TaskType1*, il task *Task2* è di tipo *TaskType2*, il task *Task3* è ancora di tipo *TaskType1*, e il task *Task4* è di tipo *TaskType3*.

Nella definizione grafica di una production le ellissi rappresentano quindi i task che la compongono, mentre i rettangoli rappresentano i loro input/output, che vengono definiti come *data box* della produzione.

Nell'esempio di figura 4.7, i data box *Input1*, *Input2*, *Input3* e *Input4* sono input della produzione.

Come descritto nel paragrafo 4.5.3, le produzioni vengono poi usate per definire

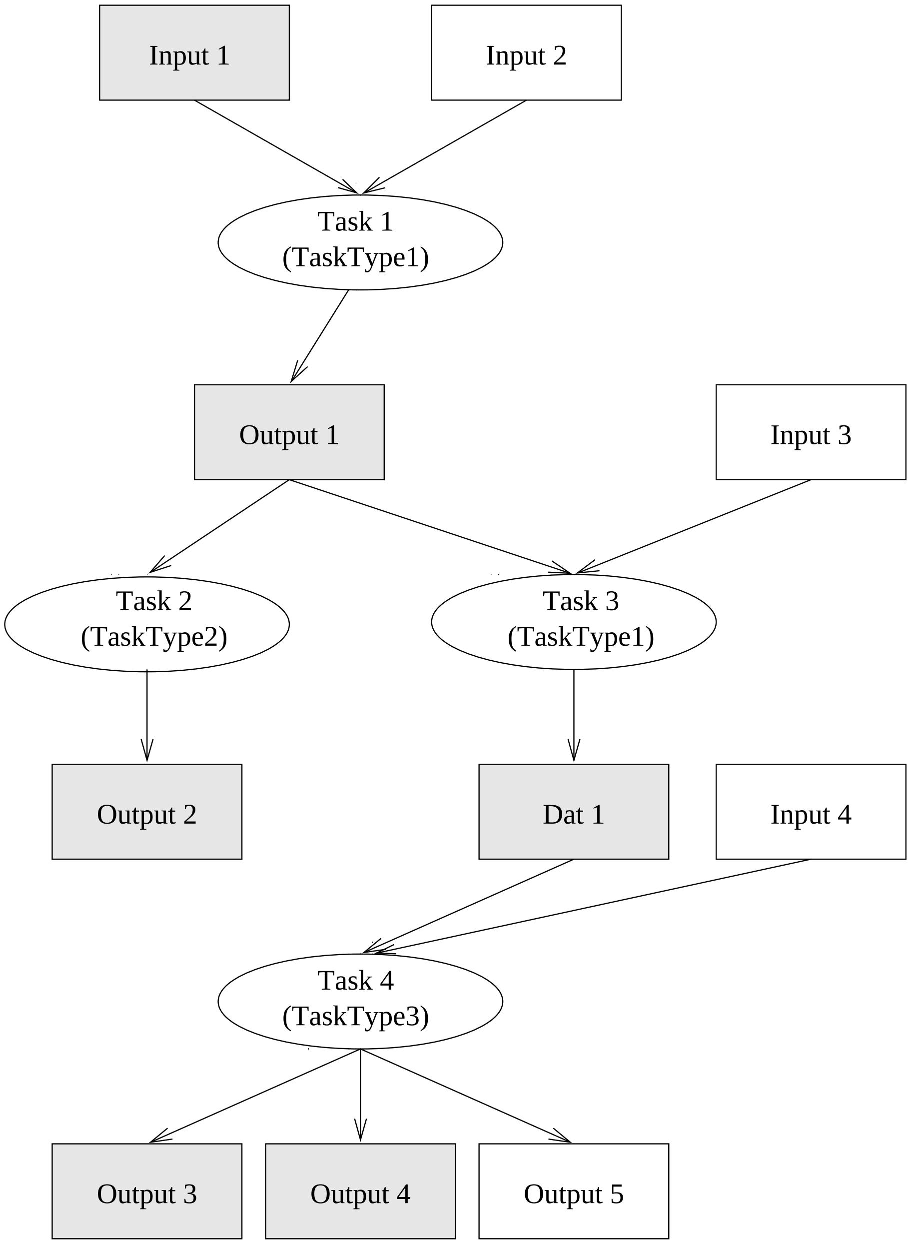


Figura 4.7: Esempio della rappresentazione grafica di una production

richieste computazionali, per elaborare specifici dati. Una produzione può essere utilizzata più volte, in momenti diversi, per elaborare dati diversi.

In base a come si "associano" i dati agli input di una produzione, questi vengono classificati in due categorie:

- input di tipo **stream**

  In una richiesta computazionale, ad un input di tipo stream di una produzione viene associata una lista di file, che devono essere elaborati uno alla volta. La collezione di questi file viene definita come *stream input set*, e ogni singolo file come *stream input file*;

- input di tipo **singleton**

  In una richiesta computazionale, ad un input di tipo singleton di una produzione viene associato un singolo file, chiamato *singleton input file*, unico elemento del *singleton input set*.

In modo analogo si possono classificare gli output della produzione (nell'esempio di figura 4.7 i data box *output1*, *output2*, *output3* e *output4*):

- output di tipo stream

  in una richiesta ad ognuno di essi è associato un insieme di file di output (*stream output set*), ognuno dei quali viene chiamato *stream output file*;

- output di tipo singleton

  in una richiesta ad ognuno di essi viene associato un singolo file (chiamato *singleton output file*).

Oltre ai data box input e output, esistono poi i data box a cui nelle richieste sono associati file "temporanei", che possono essere cancellati al termine dell'elaborazione, in quanto sono risultati intermedi che non si vogliono "salvare" (il data box *Dat1*, nell'esempio considerato).

Nella rappresentazione grafica proposta, un data box di tipo singleton è rappresentato da un rettangolo bianco, mentre uno di tipo stream è rappresentato attraverso un rettangolo "ombreggiato".

I dati di tipo stream corrispondono agli event data degli esperimenti HEP, mentre i dati di tipo singleton ai long term data.

Sempre con riferimento alla produzione di figura 4.7, si consideri una richiesta computazionale per cui all'input di tipo stream *Input1* viene associato lo stream input set costituito dalla seguente lista di stream file:

```
file1
file2
file3
file4
...
...
filen
```

mentre agli input di tipo singleton *Input2*, *Input3* e *Input4* vengono associati rispettivamente i file *singlefile2*, *singlefile3* e *singlefile4*, come rappresentato in figura 4.8.

Nell'elaborazione del task *Task1* si deve considerare il prodotto cartesiano *InpSet1* X *InpSet2*, dove *InpSet1* rappresenta lo stream input set associato all'input *Input1* della produzione, mentre *InpSet2* è il singleton input set (costituito dall'unico file *singlefile2*) associato all'input *Input2* della produzione: nella prima iterazione della computazione si deve elaborare la coppia di file *(file1, singlefile2)*, nella seconda iterazione si deve considerare la coppia *(file2, singlefile2)*, ecc... Nella prima iterazione verrà prodotto il file di output *out1file1* (per l'output di tipo stream *Output1*); nella seconda iterazione verrà generato il file *out1file2*, ecc... Per il task *Task3*, nella prima iterazione si considerano i file *out1file1* e *singlefile3* (rispettivamente associati ai data box *Output1* e *Input3*), producendo per il data box *Dat1* il file *dat1file1*; nella seconda iterazione si elaborano i file *out1file2* e *singlefile3*, generando il file *dat1file2*, ecc... Per quel che riguarda l'output *Output5*, di tipo singleton, il relativo file *outsinglefile* sarà il risultato di tutte le n iterazioni considerate.

Una production può avere:

- zero o un input di tipo stream;
- zero o più input di tipo singleton;
- zero o più output di tipo stream;
- zero o più output di tipo singleton.

Le definizioni delle production vengono memorizzate in un database, chiamato **Production Database**; ogni elemento di questo database viene descritto dai seguenti campi:

- il nome della production (es. *ProdY*);
- la descrizione (es. *Production che gestisce l'elaborazione Y*);

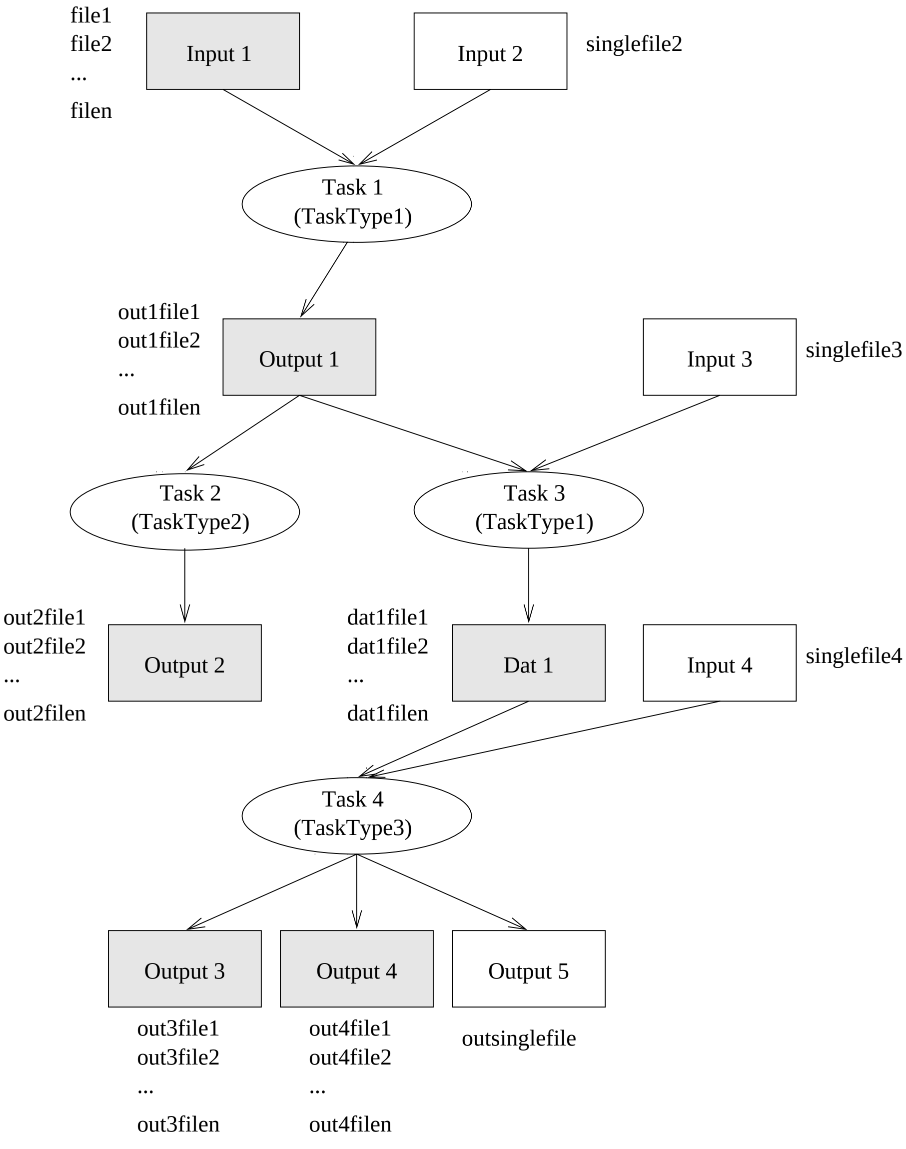


Figura 4.8: Esempio di utilizzo di una production in una computational request per l'elaborazione di dati

- lo user identifier dell'utente che ha definito la production;
- i task che compongono la production, specificandone i relativi tipi (questi tipi di task devono essere definiti nel Task Database);
- le relazioni che esistono tra i task componenti la production;
- i data box della production.

  È necessario specificare:

  - il loro numero;
  - il loro nome (es. *Input1*, *Input2*, *Dat1*, *Output1*, *Output2*, ecc...);
  - il "ruolo" di questi dati, che può essere:
    - dati di input (*InputX* nell'esempio di figura);
    - dati di output (*OutputX* nell'esempio di figura);
    - dati "temporanei" (*DatX* nell'esempio di figura);
      questi dati, una volta che non sono più necessari (sono stati usati come input da uno o più task) possono essere cancellati.
  - il loro tipo (stream o singleton);
- altre eventuali informazioni.

Oltre alla definizione di nuove production, gli utenti devono essere in grado di cancellare production precedentemente definite, di modificare informazioni relative a elementi presenti nel Production Database, di visualizzare gli elementi definiti nel database, ecc...

4.5.3 Submit di richieste computazionali

Come introdotto nel paragrafo precedente, ogni utente può sottomettere proprie richieste computazionali (computational request) al sistema DEMI, ovvero richieste di eseguire specifiche elaborazioni (rappresentate dalle production) su specifici dati, definiti nel Data Database.
Per sottomettere una computational request, l'utente deve definire:

- il nome della production da usare in questa elaborazione (es. *ProdY*).

  Questa production deve essere definita nel Production Database;

- gli input set (di tipo singleton e di tipo stream).

  Per ogni input di tipo singleton della corrispondente production è necessario specificare l'identificatore del file che deve essere usato (questo singolo file costituisce il corrispondente singleton input set).

  Per l'eventuale input di tipo stream (una production può avere al più un input di tipo stream) si deve definire il relativo stream input set, ovvero i file di dati che si devono utilizzare. Deve quindi essere specificata una lista del tipo:

  ```
  fileid1
  fileid2
  fileid3
  fileid4
  ...
  ...
  ```

  dove ogni riga rappresenta un file identifier (che deve essere definito nel Data Database).

  Se la production non ha input di tipo stream, è necessario specificare il numero di iterazioni, cioè il numero di volte in cui tale production deve essere eseguita;

- gli output (di tipo stream e di tipo singleton).

  Per ogni output della corrispondente produzione l'utente deve definire:

  – la descrizione del tipo dei file di output che saranno prodotti (es. *QQPS, 183 GeV, 97_1 MC*): questa informazione corrisponde al relativo campo del Global Data Database (vedi par. 4.4.1);

  – dove memorizzare i file che verranno generati; l'utente può scegliere di archiviare questi file di output:

    ∗ in una sua directory "privata";
    in questo caso dovrà essere specificato il pathname di questa directory;

    ∗ su nastri;
    in questo caso l'utente può definire il modello di nastri da utilizzare (modello che deve essere definito nel Tape Device Model Database). Questa opzione può comunque essere omessa, se non è richiesto l'utilizzo di uno specifico modello di nastri;

    * sui disk buffer gestiti dal sistema DEMI (vedi par. 4.3.2);

- la dimensione massima richiesta per un singolo file, associato ad ogni data box della production considerata.

  Questo parametro è necessario per stabilire quanto spazio deve essere allocato per memorizzare i file generati nei vari processing;

- un valore numerico, chiamato 'priority level', che definisce il livello di priorità per la computational request: un valore elevato di questo parametro indica la necessità di soddisfare prima possibile la richiesta;

- altre eventuali informazioni.

L'utente, quindi, non deve preoccuparsi di scegliere i file identifier per gli output che vengono prodotti. Questo viene fatto dal sistema che, una volta elaborata la richiesta, comunica all'utente gli identificatori dei file di output generati.

Tutti i dati relativi ad una richiesta vengono memorizzati in un database, chiamato **Request Database**.

Ad ogni computational request viene associato uno specifico identificatore, chiamato **REQID** (**REQuest IDentifier**), attraverso il quale l'utente può ad esempio verificare lo stato di elaborazione della richiesta, può ottenere la lista degli identificatori dei file di output prodotti fino a quel momento, ecc...

Come rappresentato in figura 4.9, il Task Database, il Production Database e il Request Database sono database "globali", di cui esiste un'unica istanza per tutto il sistema DEMI.
Nella figura sono anche schematizzate le fasi in cui si articola la definizione di una elaborazione:

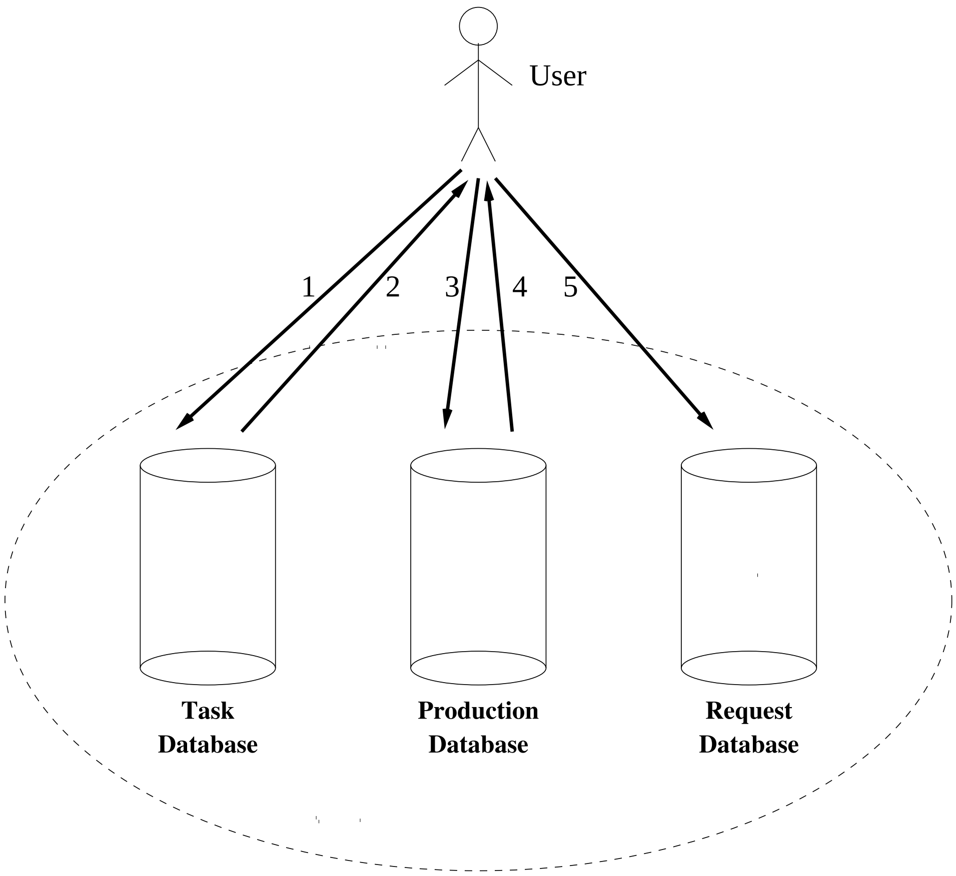
1. definizione di tipi di task;

2. selezione di tipi di task per la definizione di una production;

3. definizione di una production;

4. selezione di una production per la definizione di una computational request;

5. submit di una computational request.



1: Definizione di tipi di task

2: Selezione di tipi di task per la definizione di una production

3: Definizione di una production

4: Selezione di una production per la definizione di una computational request

5: Submit di una computational request

Figura 4.9: Definizione di una elaborazione

4.6 Elaborazione

La richiesta computazionale sottomessa dall'utente viene prima di tutto "scomposta" in **elementary request**: se la produzione corrispondente alla richiesta ha un input di tipo stream, ogni elementary request corrisponde ad un singolo stream file associato a questo input. Se invece la production non ha input di tipo stream, ogni elementary request corrisponde ad un'iterazione (il numero di iterazioni deve essere stato specificato dall'utente durante il submit della richiesta).

Si supponga, ad esempio, che la production corrispondente alla computational request considerata abbia due input di tipo singleton (*Input1* e *Input2*), e un input di tipo stream (*Input3*).

Si supponga che per il primo input di tipo singleton sia stato specificato il file:

```
singlefile1
```

mentre per il secondo input di tipo singleton sia stato specificato il file:

```
singlefile2
```

Si supponga infine che, per l'input di tipo stream, sia stato specificato come stream input set la seguente lista di file identifier:

```
fileid1
fileid2
fileid3
fileid4
...
...
fileidn
```

In questo caso la richiesta viene "scomposta" in n elementary request, a cui sono associati i seguenti file di input:

```
                 INPUT1            INPUT2          INPUT3

el-req-1        singlefile1       singlefile2     fileid1
el-req-2        singlefile1       singlefile2     fileid2
el-req-3        singlefile1       singlefile2     fileid3
el-req-4        singlefile1       singlefile2     fileid4
...             ...               ...             ...
...             ...               ...             ...
el-req-n        singlefile1       singlefile2     fileidn
```

È importante sottolineare un aspetto, relativo agli input e agli output. Ad ogni input di tipo stream di una produzione viene associato un singolo file, per ogni elementary request. Per quel che riguarda gli input di tipo singleton, invece, ad ognuno di essi viene associato un file per ogni richiesta: questo file è quindi lo stesso per tutte le elementary request che "compongono" la richiesta. Quindi ogni stream input file fa riferimento ad una singola elementary request, mentre un singleton input file è associato alla computational request. Un discorso del tutto analogo si può fare per gli output: gli stream output file sono associati alle elementary request, mentre i singleton output file sono associati alle computational request.

Le elementary request vengono inserite nella cosiddetta **global queue of elementary requests**, una struttura dati in cui l'inserimento avviene in maniera ordinata rispetto al valore del parametro 'priority level': quando una nuova elementary request deve essere inserita nella global queue of elementary requests, essa viene posta dopo le elementary request con stessa o più elevata priorità precedentemente inserite, ma prima delle elementary request a più bassa priorità.

Le elementary request memorizzate nella global queue of elementary requests sono pronte per essere elaborate.

4.7 Scheduling

4.7.1 Scheduling globale

Le elementary request memorizzate nella global queue of elementary requests devono essere "assegnate" ai vari siti del sistema DEMI, per la loro elaborazione. Obiettivo principale che deve essere considerato in questo problema di mapping globale è la massimizzazione del *throughput*, cioè la "quantità" di elaborazione complessiva realizzata dal sistema in un certo intervallo di tempo.

Il mapping delle elementary request ai siti del sistema DEMI viene affidato ad un **global scheduler**, che deve analizzare queste elementary request, considerando prima quelle memorizzate in "testa" alla global queue of elementary requests (le elementary request con più elevato valore del parametro 'priority level'), e "associarle" ai siti più adatti per la loro elaborazione.

Le regole generali che il global scheduler deve considerare sono:

- una elementary request deve essere assegnata ad un sito in grado di elaborarla.

  Un'elementary request può essere elaborata in un sito:

  - se in quel sito tutti i task componenti la corrispondente production possono essere eseguiti (ci sono computer di tutte le architetture necessarie);
  - se in quel sito ci sono i necessari tape device in grado di leggere/scrivere da/su nastro i file di input/output;

- se la produzione associata ad una computational request ha uno più output di tipo singleton, tutte le elementary request componenti questa richiesta devono essere assegnate allo stesso sito (infatti per tutte queste elementary request si deve scrivere su stessi singleton output file).

Se non è possibile assegnare una elementary request a nessun sito (ad esempio perché un task, componente della corrispondente production, è stato implementato solo per una certa architettura, e in nessun sito vi sono calcolatori di questa architettura), l'elementary request non può essere elaborata: essa viene rimossa dalla global queue of elementary requests, e l'utente che ha sottomesso la corrispondente computational request viene avvisato riguardo il problema.

Lo scheduling globale è un problema molto complesso, in cui devono essere considerati vari aspetti, ad esempio:

- la potenza computazionale disponibile nei vari siti.

  Ovviamente in questo calcolo devono essere considerati solo i computer in grado di eseguire i task delle production considerate;

- i siti dove sono fisicamente presenti i dati necessari per l'elaborazione.

  Lo spostamento di dati tra i vari siti può infatti richiedere tempo e risorse non trascurabili: la dimensione di questi file può essere infatti molto elevata.

  Come già detto, per la gestione di questo problema deve essere implementato un *data mover*, che deve decidere se, quando e come (usando la Wide Area Network, usando spedizioni via corriere, ecc...) i dati devono essere spostati da un sito ad un altro;

Per la realizzazione del global scheduler è quindi necessario definire specifici criteri (quando può essere utilizzata la Wide Area Network per la copia dei dati, qual è la forma più conveniente di file transfer, ecc...), in cui sono richiesti appositi strumenti (ad esempio per il monitoring dello stato dei link della WAN in tempo reale). Come descritto nel capitolo 6, il global scheduler non è stato implementato nel primo prototipo realizzato.

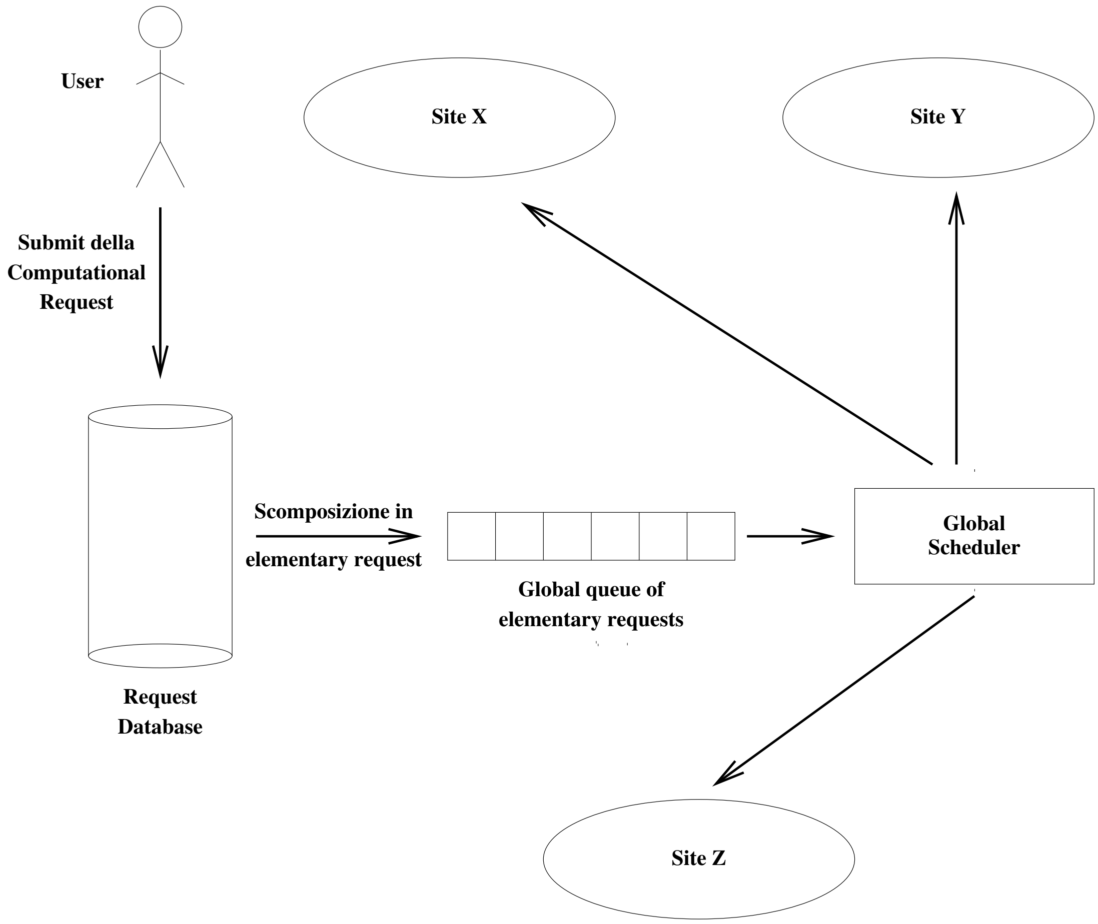


Figura 4.10: Sistema di scheduling globale

Una visione complessiva del sistema di scheduling globale è mostrata in figura 4.10: un utente, presente in un certo sito, sottomette una computational request: le relative informazioni vengono memorizzate nel Request Database. La computational request viene "scomposta" in elementary request, che vengono memorizzate nella global queue of elementary requests. Il global scheduler esamina queste elementary request, e per ognuna di esse sceglie il sito "migliore" per l'elaborazione.

4.7.2 Scheduling locale

Le elementary request assegnate ad un sito dal global scheduler, vengono memorizzate in una particolare coda, chiamata **local queue of elementary requests**. Esiste quindi una local queue of elementary requests per ogni sito del sistema DEMI.

La local queue of elementary requests ha la stessa struttura della global queue of elementary requests: le elementary request memorizzate in questa coda sono ordinate rispetto al valore del parametro 'priority level'. Quando una nuova elementary request deve essere inserita nella local queue of elementary requests, essa viene posta dopo le elementary request con stessa o più alta priorità precedentemente inserite, ma prima delle elementary request con più bassa priorità.

Le elementary request vengono analizzate, una alla volta, da un **local scheduler**, che considera per primi gli elementi memorizzati in "testa" alla local queue of elementary requests (le elementary request con più elevata priorità). Per ogni elementary request il local scheduler esamina quali sono i task che compongono la corrispondente production, e ognuno di essi viene "assegnato" ad un computer per l'elaborazione. Ovviamente l'esecuzione di un task può iniziare solo quando tutti i suoi input sono disponibili.

Si consideri ad esempio una elementary request che fa riferimento alla produzione rappresentata in figura 4.7. Per questa elementary request i file associati ai quattro input (*Input1*, *Input2*, *Input3* e *Input4*) siano rispettivamente *file1*, *singlefile2*, *singlefile3* e *singlefile4*, come rappresentato in figura 4.11 (i file *singlefile2*, *singlefile3* e *singlefile4* saranno usati anche per le altre elementary request della stessa computational request, trattandosi di singleton input file).

Il task *Task1* può iniziare subito l'esecuzione, considerando i file di input *file1* e *singlefile2*: verrà prodotto lo stream output file *out1file1*. Il task *Task2* può iniziare l'esecuzione solo dopo che il task *Task1* è terminato: come input viene considerato il file *out1file1*, e viene generato lo stream output file *out2file1*. Anche il task *Task3* può essere eseguito solo dopo il completamento dell'elaborazione del task *Task1*: usando i file di input *out1file1* e *singlefile3*, viene prodotto lo stream file *dat1file1*. Infine il task *Task4*, che utilizza come input i file *dat1file1* e *singlefile4*, può essere eseguito dopo l'esecuzione del task *Task3*. Dopo questa elaborazione, lo stream file temporaneo *dat1file1* può essere cancellato.

Obiettivo di questo problema di mapping locale è la massimizzazione del throughput, cioè la "velocità" complessiva con cui i dati di input vengono elaborati. Non sono invece posti vincoli riguardo la *latenza* (cioè il tempo necessario per l'elaborazione di un singolo input).
A tal scopo il local scheduler deve considerare vari parametri, quali:

- i requisiti di eterogenità.

  Un task può essere stato implementato solo per un certo numero di architetture; dovrà quindi essere scelto un calcolatore in grado di eseguire questa elaborazione;

- il tipo di architettura dei computer.

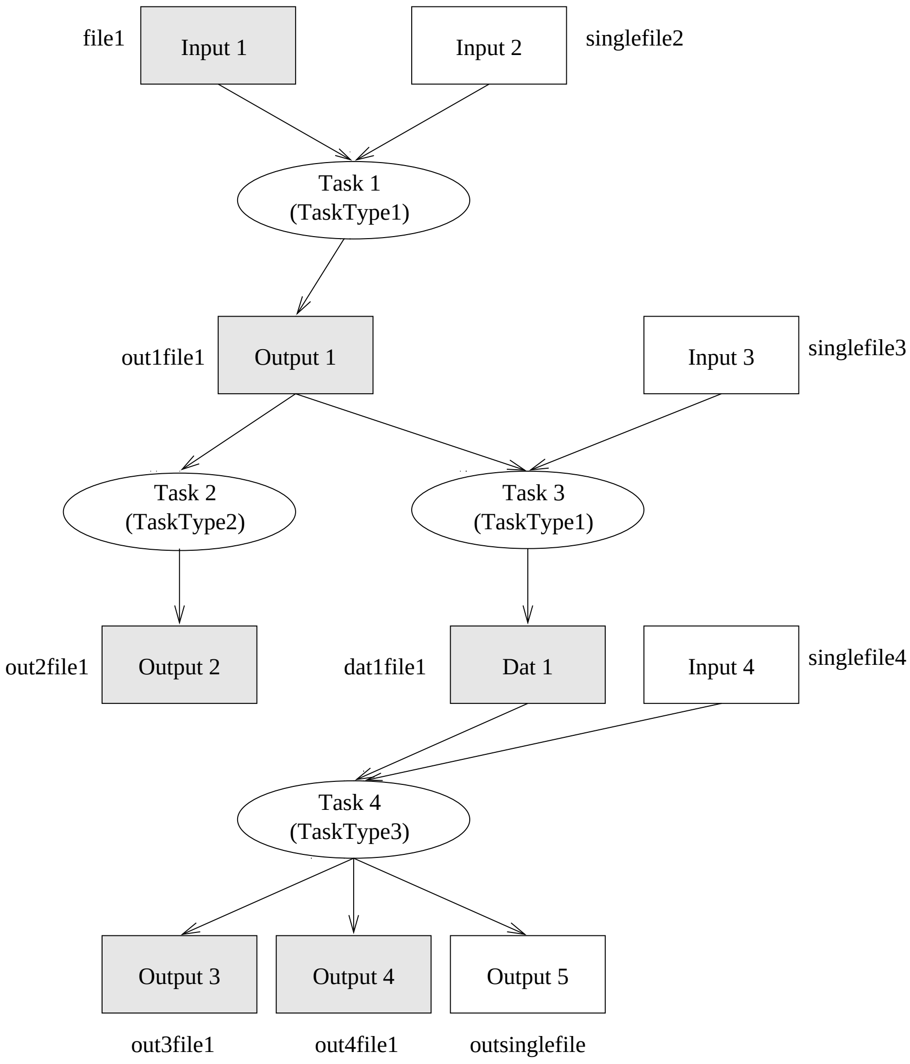


Figura 4.11: Elaborazione di una elementary request

Un task può essere stato implementato per più architetture, ma possono esserci architetture più adatte per l'elaborazione. Per questo scopo si considera l'attributo 'cost of the task' del Task Database;

- le performance dei computer.

  I vari computer possono avere diverse prestazioni: a tal fine il local scheduler deve considerare il parametro 'performances' del Local Computer Database;

- il *load* dei vari computer presenti nel sito, considerando anche quello non dovuto a job DEMI (job sottomessi da altri utenti).

Non è invece necessario considerare il problema della conversione dei dati, tra calcolatori di diversa architettura, e che quindi possono usare metodi diversi per la loro rappresentazione: si può infatti supporre che in tutti i computer si usino gli stessi formati standard per gestire i dati.

In generale un task non può essere in esecuzione su un computer che in quel momento è usato da un utente generico. Se un task DEMI inizia la sua esecuzione su un calcolatore che poi viene "reclamato" da un utente, l'esecuzione del task DEMI deve essere sospesa (o, eventualmente, il job può essere migrato su un altro calcolatore disponibile): l'elaborazione può essere ripresa solo quando l'utente ha terminato di usare tale computer. Questa "politica di scheduling" può comunque essere diversa tra i vari siti: ad esempio si potrebbe considerare un ambiente in cui le workstation devono essere utilizzate solo per il sistema DEMI, che quindi deve avere più elevata priorità rispetto ai processi interattivi degli utenti.

Quando un'elementary request è stata elaborata con successo, il sistema deve aggiornare il Data Database, per registare i nuovi stream output file che sono stati prodotti.

Quando è terminata l'elaborazione di una richiesta computazionale (cioè tutte le "sue" elementary request sono state elaborate), il Data Database deve essere aggiornato con i relativi singleton output file. L'utente che aveva sottomesso tale richiesta deve essere avvertito, e gli deve essere comunicata la lista degli identificatori dei file di output generati.

Una visione del sistema di scheduling locale è presentata in figura 4.12: le elementary request assegnate ad un certo sito dal global scheduler vengono memorizzate nella local queue of elementary requests. Il local scheduler esamina queste elementary request, e sceglie i computer più adatti dove eseguire i corrispondenti task.

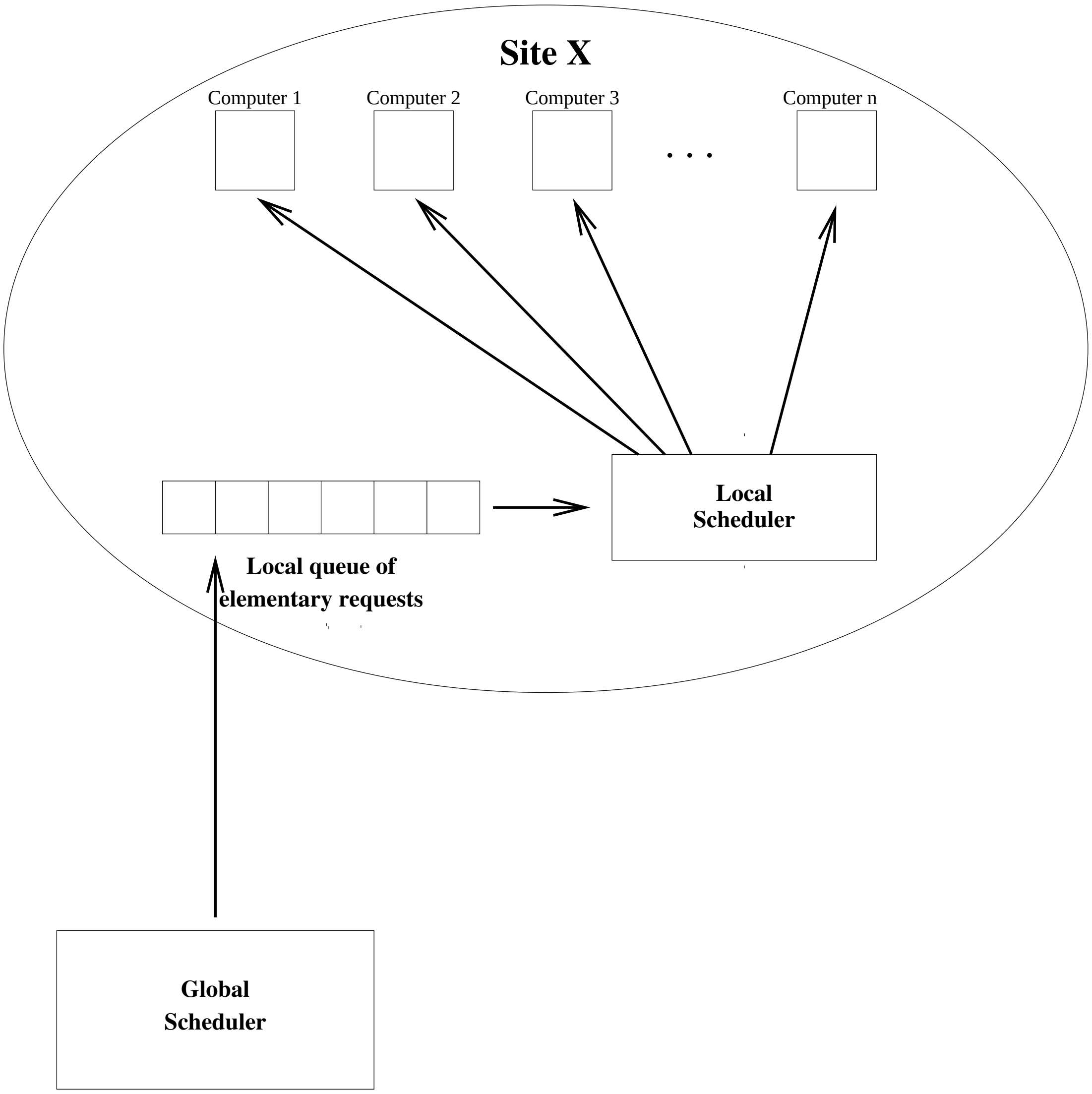


Figura 4.12: Sistema di scheduling locale

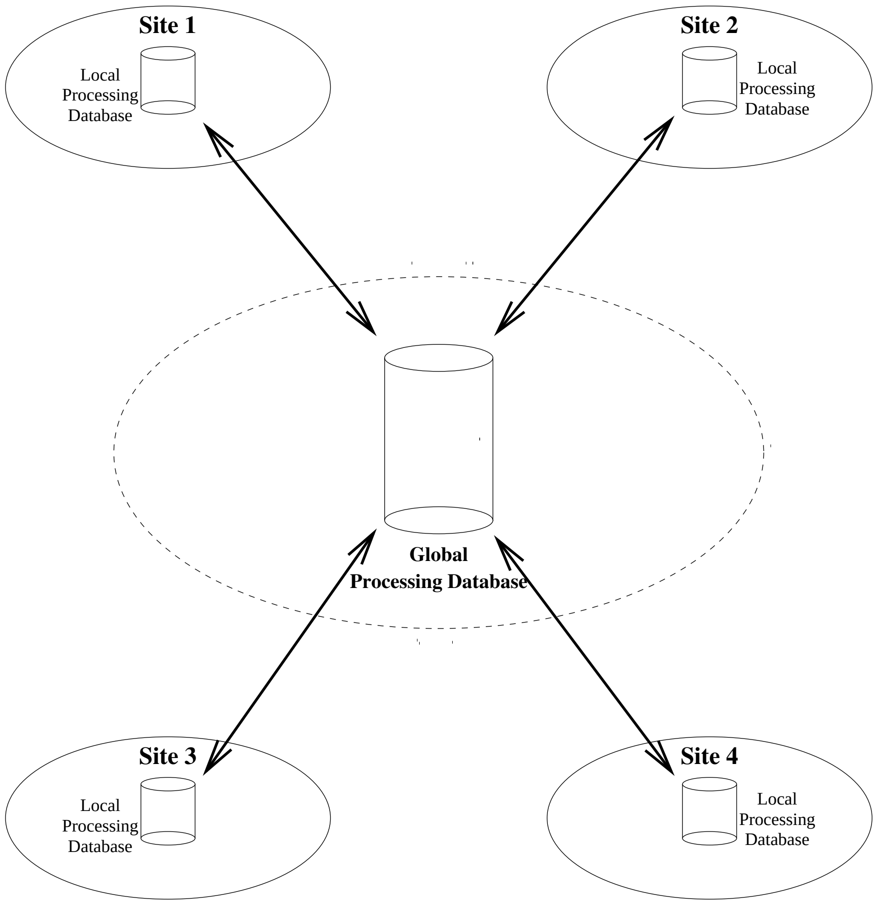


Figura 4.13: Processing Database

4.8 Processing Database

Per tener traccia di tutte le elaborazioni eseguite nei diversi siti del sistema DEMI, deve essere utilizzato uno specifico database, chiamato **Processing Database**.

La struttura del Processing Database è mostrata in figura 4.13: esso consiste quindi di un **Global Processing Database** "centrale" e, in ogni sito, di un **Local Processing Database**.

4.8.1 Global Processing Database

Ogni elemento del Global Processing Database rappresenta un **elementary processing**, ovvero l'elaborazione di una elementary request.

Un elementary processing è rappresentato nel Global Processing Database dai seguenti campi:

- l'identificatore della elementary request (**elementary request identifier**);
- il REQID (REQuest IDentifier) della corrispondente richiesta computazionale;
- il sito dove il processing è stato eseguito;
- lo stato dell'elaborazione (in esecuzione, elaborazione completata con successo, processing fallito, ecc...);
- altre eventuali informazioni.

4.8.2 Local Processing Database

In ogni sito del sistema DEMI è presente un Local Processing Database. Ogni elemento di questo database rappresenta l'elaborazione di una elementary request eseguita in quel sito, ed è descritto dai seguenti campi:

- l'elementary request identifier;
- per ogni elaborazione di ogni task componente la corrispondente production, deve essere specificato:
  - il nome del task;
  - il computer dove il processing è stato eseguito;
  - la data e l'ora in cui è iniziata l'elaborazione;
  - la data e l'ora in cui il processing è terminato;
  - lo stato dell'elaborazione (in esecuzione, elaborazione completata con successo, processing fallito, ecc...).

  È importante notare come, per uno stesso singolo task relativo ad una elementary request, potrebbero essere effettuate più elaborazioni: infatti un processing potrebbe non essere stato completato con successo (ad esempio per un crash del computer), e quindi eseguito ancora;
- altre eventuali informazioni.

Si supponga, ad esempio, che un'elementary request sia stata elaborata nel sito *SiteX*. Nel Global Processing Database deve essere presente un elemento relativo a questo elementary processing, in cui è appunto indicato che l'elaborazione è avvenuta nel sito *SiteX*. Nel Local Processing Database, implementato nel sito *SiteX*, dovrà essere presente una entry relativa a questo processing, a cui saranno associate tutte le corrispondenti informazioni.

Quindi nel Global Processing Database vi è un elemento per ogni elementary processing gestito dal sistema DEMI, mentre in ogni Local Processing Database c'è un elemento per ogni elementary processing effettuato nel corrispondente sito.

Gli utenti del sistema DEMI devono poter essere in grado di effettuare ricerche di elementi del Processing Database, utilizzando in questa query uno o più campi. Il sistema deve aggiornare automaticamente i vari attributi del Processing Database dopo ogni processing: non deve essere il singolo utente ad eseguire manualmente queste operazioni.

4.9 Logbook

In ogni sito del sistema DEMI viene utilizzato un **Local Logbook**, per tener traccia di tutti gli eventi significativi e di tutte le condizioni di errore avvenuti nel sito.
Ogni elemento di questo database è rappresentato dai seguenti campi:

- **categoria**;

  questo campo può essere uguale a:

  - 'info';

    definisce un evento "normale" occorso nel sistema;
  - 'error';

    definisce una condizione di errore;

- **tipo**;

  una sorta di "titolo", che descrive brevemente l'evento (es. *NewComputer*);

- **descrizione**;

  la descrizione "completa" dell'evento/errore (es. *Il local administrator X ha aggiunto al sito il computer Y*);

- **data**;
  la data in cui si è verificato l'evento;
- **ora**;
  l'ora in cui si è verificato l'evento.

Oltre ai Local Logbook, deve essere presente un **Global Logbook** "centrale", usato per memorizzare le informazioni relative agli eventi significativi e agli errori avvenuti nel sistema a livello globale. Il Global Logbook ha esattamente la stessa struttura del Local Logbook.

Il Global Logbook deve essere gestito unicamente dal global administrator, mentre i Local Logbook sono gestiti solamente dai corrispondenti local administrator. Gli amministratori possono effettuare su questi logbook operazioni di ricerca, di cancellazione di elementi, ecc...

Capitolo 5

Strumenti disponibili

Anche se è ormai convinzione comune che è in genere più conveniente utilizzare negli ambienti di calcolo un insieme di workstation, anziché un singolo potente supercalcolatore centrale, ci sono stati (ed esistono tuttora) diversi ostacoli per l'affermarsi di questo modello di calcolo distribuito.

Il problema più serio è probabilmente la difficoltà nello sviluppare programmi di tipo parallelo, adatti a questo tipo di ambiente. È infatti molto più complesso scrivere programmi paralleli, rispetto a programmi puramente sequenziali: il programmatore deve gestire direttamente diversi dettagli implementativi, quali la sincronizzazione e la comunicazione tra i vari processi che costituiscono il programma, il mapping dei processi ai processori disponibili, ecc...

Per semplificare lo sviluppo di applicazioni di questo tipo, e la corrispondente esecuzione in un sistema di calcolo distribuito, negli ultimi anni sono stati sviluppati appositi ambienti visuali. Alcuni di questi vengono esaminati in appendice B, con particolare riferimento alle caratteristiche e funzionalità richieste per il sistema DEMI. Allo stato attuale non esiste, purtroppo, nessun ambiente che soddisfi tutti i requisiti necessari per il sistema da realizzare.

In questo capitolo si pone l'attenzione su un altro tipo di strumenti, utili per gestire le elaborazioni in un sistema costituito da decine, centinaia, o migliaia di workstation. Come spiegato in [34], considerando un ambiente di questo tipo, è molto probabile che si verifichino situazioni per cui ci sia almeno un calcolatore "libero", mentre molti job sono in attesa di essere eseguiti su altre workstation. Il tempo di risposta può essere, quindi, molto alto, anche se la "potenza computazionale" complessiva del sistema è sottoutilizzata.

Il problema può essere gestito in maniera adeguata con l'impiego di sistemi di load sharing, in seguito descritti.

5.1 Sistemi di load sharing

Visti i prezzi sempre più competitivi, le workstation e i personal computer sono ormai ovunque: considerando un qualsiasi ambiente del mondo scientifico o industriale, praticamente ogni utente ha il suo calcolatore personale.

Il problema è come poter sfruttare al meglio questi computer, che spesso hanno performance molto elevate. Usare, ad esempio, un'alpha station o una workstation HP solo per leggere la posta elettronica o per scrivere documenti, non significa certo raggiungere il limite della potenza del calcolatore. Inoltre, spesso questi computer sono del tutto inutilizzati dai corrispondenti "proprietari": di notte, nei weekend, all'ora di pranzo, quando ci sono riunioni, ecc... D'altro canto ci sono applicazioni che richiedono enormi risorse computazionali, per cui l'utilizzo di una singola workstation è insufficiente (o sarebbe comunque richiesto un tempo per l'esecuzione troppo elevato).

Esistono diversi sistemi di load sharing in grado di gestire questi tipi di problemi. Un *sistema di load sharing* è un sistema software utilizzato per lo scheduling di job su un insieme di workstation connesse tra loro in rete. Attraverso questo tipo di applicazioni, le varie workstation vengono viste dall'utente come un unico grande sistema di calcolo: è compito del sistema di load sharing "assegnare" i job ai computer disponibili, in modo da garantire un utilizzo efficiente delle risorse.

Come detto, nel sistema DEMI si vuole garantire l'utilizzo ottimale delle workstation, per l'esecuzione dei vari job. Per questo motivo, e per poter gestire le diverse configurazioni (nei vari siti può essere necessario considerare diverse "politiche" per la gestione di questi calcolatori: computer "dedicati" al sistema DEMI, computer di cui si possono usare solo i cicli macchina liberi, ecc...) è necessario poter utilizzare le funzionalità offerte da un load sharing system.

Sono attualmente disponibili diversi sistemi di load sharing: tra questi citiamo *Condor*, *LSF*, *Codine* [26], *DQS* [55], *IBM LoadLeveler* [29], *NQE* [21] e *NQS*. Alcuni di essi sono utilizzabili gratuitamente, altri invece (come LSF, IBM LoadLeveler, Codine e NQE) sono prodotti commerciali, che vanno quindi regolarmente acquistati.

Vengono di seguito descritti due tra i sistemi di load sharing più interessanti: Condor e LSF.

5.1.1 Condor

Condor [16] [32] [33] [36] [46] è un sistema di load sharing sviluppato da un team di ricercatori del Computer Sciences Department, University of Wisconsin, Madison, USA, diretto dal prof. Miron Livny. La prima versione del sistema risale a circa dieci anni fa; da allora sono state introdotte nuove funzionalità e supportate nuove architetture: attualmente esistono implementazioni di Condor per molte piattaforme Unix (Sun SunOS, Sun Solaris, DECStation MIPS Ultrix, DEC Alpha Digital Unix, HP HPUX, Linux). È attualmente in corso il porting a Windows NT, grazie anche ad un finanziamento da parte di Microsoft.

I file binari del sistema Condor e i relativi sorgenti sono disponibili gratuitamente.

Condor è in grado di gestire solamente i processing di batch job: non c'è supporto per elaborazioni interattive.

L'osservazione fondamentale su cui si basa il sistema Condor è che in un ambiente di lavoro, di tipo industriale o accademico, ogni persona "proprietaria" di un computer in genere utilizza tale risorsa solo per una frazione di tempo limitata. Si pensi ad esempio alle workstation utilizzate per applicazioni di CAD: si tratta di computer che hanno prestazioni molto elevate, ma che vengono usati solamente di giorno, per un'attività interattiva. Vi sono però utenti che hanno necessità di eseguire elaborazioni molto complesse, che richiedono una potenza di calcolo molto elevata: i loro computer non sono sufficienti per tutte queste computazioni.

Il sistema Condor gestisce questo tipo di problemi: ogni computer viene inserito in un *Condor pool* (che può essere costituito anche da calcolatori di diversa architettura), in modo tale che ogni utente "doni" i cicli macchina liberi del proprio computer agli altri utenti che ne hanno necessità.

È il sistema che sceglie il calcolatore su cui eseguire un processo: un batch job viene eseguito su un computer quando questo è "libero" (cioè quando il corrispondente utente non lo sta usando). Se durante l'esecuzione l'utente "reclama" l'uso del computer, il processing deve essere sospeso: l'elaborazione può riprendere solamente quando il calcolatore ridiventa nuovamente libero. È anche possibile decidere che il processo venga "migrato" su un altro computer.

L'amministratore del Condor pool definisce, attraverso espressioni logiche, quando un batch job può essere eseguito su un computer, quando un job deve essere sospeso (espressione di 'suspend'), quando l'esecuzione può riprendere (espressione di 'resume'), quando un job deve essere "migrato" su un altro computer (espressione di 'migrate'). Il sistema è configurabile a seconda delle diverse esigenze: in queste espressioni logiche possono essere usati parametri statici (es. l'architettura dei calcolatori, il tipo di processore, la memoria presente, ecc...) e due parametri dinamici: l'utilizzo di cpu, e il tempo di inattività interattiva del calcolatore.

Gli utenti sottomettono batch job al sistema attraverso script che definiscono quali sono le risorse necessarie per l'esecuzione. Ogni utente ha una propria priorità, che riflette l'uso passato delle risorse del pool: più si utilizza il sistema, più bassa diventa la priorità.

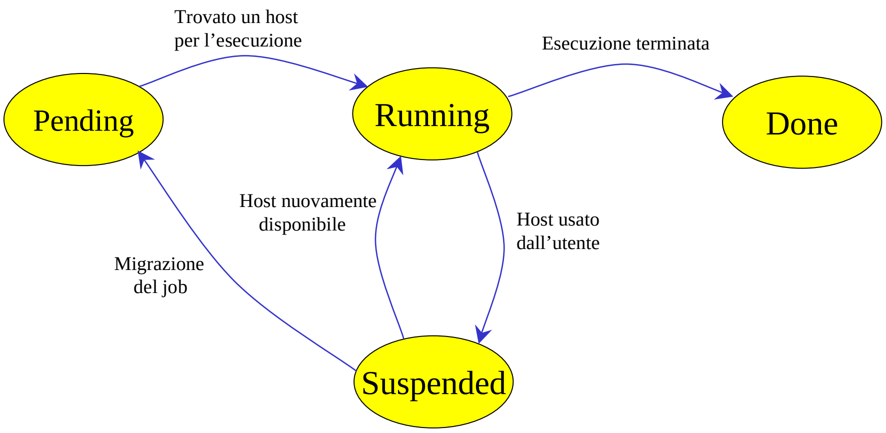


Figura 5.1: Stati di un batch job

Un job sottomesso può iniziare l'esecuzione quando vi è almeno un computer disponibile che disponga delle risorse richieste (nel caso ci siano due o più calcolatori in grado di eseguire il job, si sceglie quello che sta eseguendo un numero minore di processi).

Se durante l'esecuzione l'espressione di 'suspend' diventa vera (ad esempio perché il "proprietario" del computer riprende ad usare tale calcolatore), il job viene sospeso. Se è soddisfatta anche l'espressione di 'migrate', viene fatto un *checkpoint* del job, cioè viene fatta una copia su disco dell'immagine del job in esecuzione. A partire da questo checkpoint, il job può riprendere l'esecuzione su un altro computer, ovviamente della stessa architettura. Se invece il job non deve essere "migrato" su un altro computer, si attende finché l'espressione booleana di 'resume' diventa vera, in modo da poter riprendere l'esecuzione.

In figura 5.1 sono mostrati i possibili stati di un batch job.

Il meccanismo di checkpointing può essere utilizzato anche come sistema di fault tolerance; eseguendo periodicamente il checkpoint di un job, in caso di crash del computer dove il job è in esecuzione, il lavoro fatto fino a quel momento non viene perso: l'esecuzione può riprendere a partire dall'ultimo checkpoint.

Per quel che riguarda l'architettura del sistema Condor, una particolare workstation del pool, chiamata *Central Manager*, tiene traccia dello stato delle varie

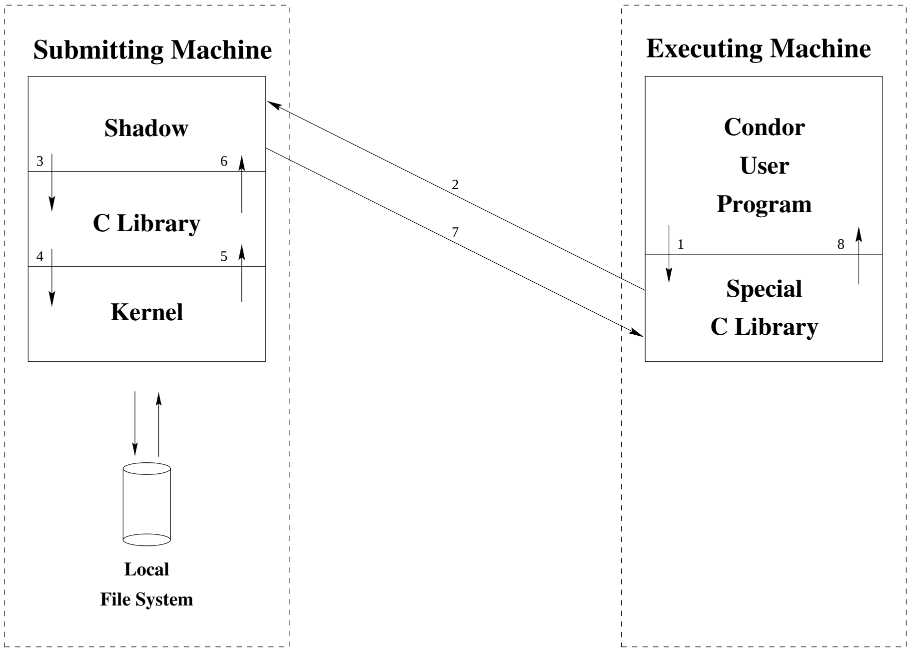


Figura 5.2: Remote system call

risorse e dei vari job. Su ogni workstation sono sempre in esecuzione due processi daemon: il processo *Startd* e il processo *Schedd*. Il processo Startd determina lo stato della workstation, verificando se essa è disponibile per eseguire job: queste informazioni vengono trasmesse al Central Manager. Il processo Startd, inoltre, gestisce l"'avvio" dell'esecuzione dei job sul computer locale, su indicazione del Central Manager. Il processo Schedd, invece, analizza quali sono i job sottomessi dagli utenti, e comunica queste informazioni al Central Manager.

Il Central Manager è il componente fondamentale del sistema: decide infatti dove i job devono essere eseguiti, in base alle informazioni comunicategli dai vari processi Schedd e Startd. Se la workstation che funge da Central Manager non è disponibile, l'attività dell'intero Condor pool è sospesa.

Non è richiesto che ogni utente abbia un account su tutte le workstation del pool, e non è nemmeno necessario che ci sia un file system comune tra tutti questi computer: un job Condor, infatti, viene eseguito come se l'ambiente di esecuzione fosse quello del computer dove il job è stato sottomesso. In figura 5.2 è mostrato come questo viene realizzato.

Il job in esecuzione sulla "executing machine" usa un'apposita versione della C library: le system call standard vengono "sostituite" da system call remote, eseguite da un processo "shadow" in esecuzione sulla "submitting machine", cioè sul computer dal quale il job è stato sottomesso. Il risultato di queste system call viene poi inviato alla macchina che sta eseguendo il job. Il processo shadow è del tutto trasparente all'utente.

Il codice dei programmi che vengono eseguiti attraverso Condor non deve essere modificato. Se si utilizza il meccanismo del checkpointing vi sono però delle limitazioni: non è possibile usare system call che creano altri processi (es. *fork*, *exec*), non è supportata la gestione dei signal, e non è possibile utilizzare meccanismi di interprocess communication (IPC).

Per quel che riguarda il linking dell'applicazione, si deve utilizzare la particolare versione della C library e, nel caso si utilizzi il meccanismo del checkpointing, un'altra specifica libreria.

Non esiste un'API che permetta di scrivere scrivere applicazioni che utilizzano direttamente le funzionalità del sistema Condor. Esiste però un'API, chiamata *CARMI* [47], che permette di sviluppare programmi PVM da eseguire poi in un Condor pool.

È anche possibile configurare un sistema costituito da diversi Condor pool, distribuiti geograficamente tra diversi siti: si parla in questo caso di *Condor flocking* [23].

5.1.2 LSF

LSF (Load Sharing Facility) [43] [44] [57] [61], precedentemente chiamato *Utopia*, è un sistema di load sharing, inizialmente sviluppato presso l'Università di Toronto, e ora venduto e supportato da Platform Computing Corporation. LSF è attualmente implementato per diversi sistemi operativi Unix (Digital Alpha, HP-UX/HPPA, IBM AIX, Linux, SGI IRIX, SunOS/Solaris, ecc...), e per Windows NT.

Attraverso questo sistema di load sharing viene gestita l'esecuzione di job, interattivi e/o batch, su un *cluster*, ovvero un insieme di computer, che possono anche essere di architetture e modelli diversi. A seconda delle condizioni di load dei vari calcolatori, e a seconda delle risorse richieste per l'applicazione (che devono essere specificate dall'utente), viene scelto per l'esecuzione il computer più adatto.

L'ambiente tipico su cui LSF opera è quello di un cluster in cui ogni utente ha un account su ogni computer, e dove vi è un file system comune. Se un utente non ha un account su tutti i computer, è possibile definire delle "associazioni" (es. si specifica che l'utente *X* dell'host *Y* corrisponde all'utente *Z* dell'host *W*). Se non c'è un file system comune tra tutte le workstation del cluster, è possibile utilizzare le apposite facility fornite dal sistema LSF per la copia di file da/a una macchina remota, per assicurarsi quindi che questi file siano presenti nel file system della workstation che esegue il job.

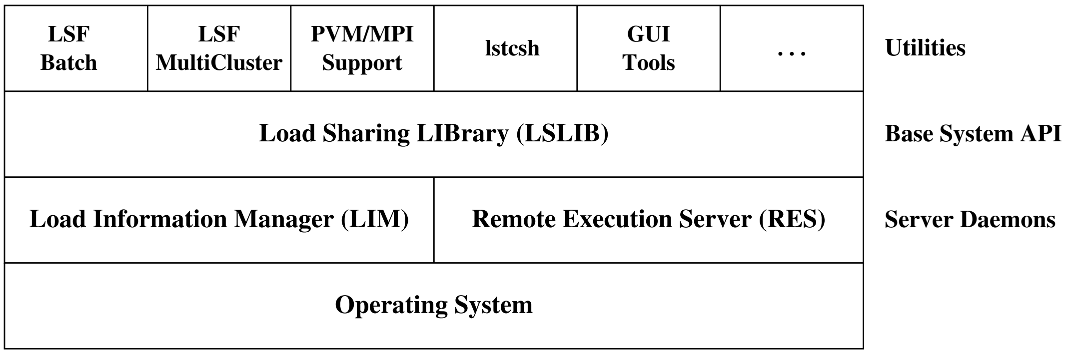


Figura 5.3: Architettura del sistema LSF

L'amministratore del cluster ha il pieno controllo sull'utilizzo dei vari computer: può decidere chi può usare un certo calcolatore e quando, può decidere quanti job di un utente (o di un gruppo di utenti) possono essere in esecuzione simultaneamente, può definire le condizioni per cui un computer può eseguire job, e le condizioni per cui i job devono invece essere sospesi, può imporre dei limiti sull'utilizzo delle risorse (cpu, memoria, ecc...) da parte dei vari job, ecc...

Per definire le condizioni di esecuzione/suspend, si possono considerare sia parametri statici (ad esempio l'architettura dei calcolatori, il loro numero di processori, la memoria presente, ecc...), che dinamici (ad esempio l'utilizzo di cpu, la memoria disponibile, l'attività di paging, l'attività di I/O, lo spazio swap disponibile, il numero di utenti che stanno usando il computer in maniera interattiva, il tempo di inattività interattiva, indici definibili dall'amministratore, ecc...).

Anche in LSF è stato implementato il meccanismo di checkpointing. In alcuni sistemi (come ConxexOS, Cray UNICOS e SPP-UX) questo viene realizzato a livello di kernel: è il sistema operativo che gestisce direttamente il checkpointing, che è del tutto trasparente all'applicazione. Nella maggior parte dei sistemi, invece, il checkpointing è implementato come "user level"[1]: nel linking dell'applicazione che deve poi essere eseguita, si deve utilizzare anche un'apposita libreria.

Non è comunque mai necessario modificare il codice del programma, e vi sono le stesse limitazioni, viste per Condor, per i programmi che utilizzano il checkpointing.

I possibili stati dei job, e le "migrazioni" tra i diversi stati, sono gli stessi di quelli visti per il sistema Condor.

L'architettura del sistema LSF è mostrata in figura 5.3.

In ogni host del cluster sono presenti due daemon: il *Load Information Manager*

[1]Il sistema di checkpointing "user level" di LSF si basa su quello implementato nel sistema Condor

(LIM) e il *Remote Execution Server* (RES): questi due processi forniscono agli utenti un ambiente uniforme, indipendente dall'architettura e dal sistema operativo dei computer.

Il processo LIM si occupa del monitor dei vari parametri del computer, e scambia queste informazioni con gli altri LIM. Tra i vari processi LIM, uno svolge le funzioni di *master*: suo compito è quello di decidere su che computer eseguire i vari job. Se vi è un crash del computer dove è in esecuzione il master, un altro LIM viene "eletto" master: il sistema LSF è quindi nel suo complesso attivo anche se solo un computer del cluster è disponibile. Il processo RES, invece, gestisce le esecuzioni dei job sottomessi da macchine remote.

La *Load Sharing LIBrary* (LSLIB) è l'Application Programming Interface (API) di LSF, che permette alle varie applicazioni di utilizzare i servizi dei processi LIM e RES. È anche possibile utilizzare direttamente questa libreria per scrivere programmi che utilizzano le funzionalità del sistema LSF [45].

Esistono poi varie applicazioni e utility, che operano attraverso chiamate a funzioni LSLIB. Tra queste applicazioni vi è la *lstcsh*, una particolare versione della shell tcsh, che permette di eseguire comandi unix o programmi sugli altri computer del cluster: il sistema LSF sceglie per l'esecuzione il calcolatore più adatto, gestendo l'I/O in maniera del tutto trasparente per l'utente.

Altre utility sono una versione load sharing di Make (che permette la compilazione/linking usando più computer contemporaneamente), tool per la gestione di job che utilizzano i sistemi di message passing PVM o MPI, un'interfaccia grafica per poter accedere facilmente alle varie funzionalità del sistema, ecc...

L'applicazione più importante è comunque *LSF Batch*, un batch system distribuito che fornisce servizi di scheduling di job sui calcolatori del cluster. L'amministratore può definire diverse code, con associati vari parametri: i computer del cluster da utilizzare, gli utenti che possono usare questa coda, la priorità rispetto alle altre code, il valore di nice con cui vengono eseguiti i job, le condizioni che devono essere soddisfatte affinché i job di questa coda possano iniziare l'esecuzione, debbano essere sospesi, debbano essere migrati su altri computer, ecc... Ad esempio, potrebbe essere definita una coda per job ad alta priorità, ma che richiedono un tempo di esecuzione limitato (definendo quindi un tempo massimo di CPU), potrebbe essere creata una coda per job a bassa priorità, che devono essere eseguiti su un computer solo se non ci sono altri job in esecuzione e se non c'è nessun utente che sta lavorando interattivamente su questo calcolatore, ecc...

Quando un utente sottomette un job ad una certa coda, questo resta in attesa finché non si rende disponibile un host adatto per l'esecuzione.

Esiste un'apposita applicazione della "suite" dei prodotti LSF, chiamata *LSF*

MultiCluster, che permette di definire relazioni tra diversi cluster LSF (l'equivalente del Condor flocking).

5.1.3 Conclusioni

I due sistemi considerati hanno caratteristiche e funzionalità diverse. Da questo confronto, LSF appare come un sistema più "evoluto" e completo rispetto a Condor.

In LSF, ad esempio, c'è il supporto per elaborazioni interattive, non previsto invece in Condor. LSF, inoltre, è più sofisticato, e meglio "configurabile": si possono definire code per specifiche esigenze, in cui si possono considerare molti parametri statici e dinamici (mentre in Condor i soli due parametri dinamici a cui si può fare riferimento sono l'utilizzo di CPU, e il tempo di inattività del computer), esistono implementazioni per un numero superiore di piattaforme (in particolare c'è il supporto anche per Windows NT), ecc... Anche per il requisito di fault tolerance, LSF è sicuramente preferibile a Condor. LSF prevede poi la possibilità di poter usare un'API, per poter sviluppare programmi che usano direttamente le funzionalità del load sharing system: questo non è possibile in Condor, a meno che non si considerino applicazioni PVM.

L'unica vera funzionalità implementata in Condor e non presente in LSF, è la possibilità di operare in un ambiente in cui non c'è un file system comune tra i vari calcolatori, senza necessità di spostare i file.

C'è da dire, infine, che LSF, a differenza di Condor, è un prodotto commerciale, con i relativi vantaggi (migliore affidabilità, possibilità di avere supporto in caso di problemi), e svantaggi (il costo).

Per il sistema DEMI si ritengono indispensabili alcune funzionalità proprie di un sistema di load sharing. Innanzittutto è necessario il supporto per la gestione di batch job, mentre non è richiesta la gestione di elaborazioni interattive. Si deve poi poter configurare il sistema (potendo considerare il numero maggiore possibile di parametri) per definire quando un job può iniziare l'esecuzione su un computer, e quando deve invece essere sospeso. Il checkpointing e la migrazione dei job possono essere considerati come funzionalità auspicabili, ma non ad alta priorità, come la possibilità di poter operare in un ambiente dove non c'è un file system comune.

A proposito del checkpointing, bisogna sottolineare che salvare su disco lo stato dei vari job ad intervalli regolari, può richiedere tempi piuttosto elevati. Oltre ad un impatto negativo sulle performance complessive, è poi necessario prevedere uno spazio disco adeguato per poter memorizzare queste immagini dei processi.

Per quel che riguarda le piattaforme supportate, deve essere possibile usare sistemi Unix (l'insieme più grande possibile), e PC Windows NT. È poi indispensabile poter utilizzare un'API, per poter scrivere applicazioni che usano le funzionalità del batch system.

Implementare un nuovo sistema che realizzi tutte queste funzionalità richiederebbe molto tempo e molte risorse (è molto complesso gestire problemi quali il load balancing, il suspend e il resume dei processi, ecc...), e comunque sarebbe un'operazione inutile, visto che esistono già sistemi che soddisfano a questi requisiti in maniera più che adeguata.

Come descritto nel capitolo 8, nella prima implementazione del sistema DEMI si sono usate le funzionalità del sistema LSF, che al momento appare come il load sharing system più completo.

Capitolo 6

Gestione dei dati

In questo capitolo e nel successivo viene presentata la fase di progettazione del sistema DEMI, cioè l'insieme delle attività che consentono di "passare" dalla specifica dei requisiti alla effettiva realizzazione del sistema.

Non viene discussa la realizzazione del global scheduler: questo modulo verrà progettato e implementato solo in una seconda fase, quando saranno meglio definiti i requisiti da considerare. Quindi nel primo prototipo si è considerato un ambiente costituito da un unico sito.

In questo capitolo in particolare si descrive il problema del trattamento delle varie informazioni, che devono essere gestite in maniera persistente. Viene presentata la progettazione dello schema concettuale di una base di dati, utilizzando il modello a oggetti, che si presta bene per "modellare" l'ambiente considerato. Viene poi descritta la trasformazione di questo schema in uno schema logico relazionale, per permettere l'implementazione della base di dati in un sistema di questo tipo.

6.1 Ambiente considerato

Nella realizzazione del primo prototipo si è considerato un ambiente costituito da un unico sito, come rappresentato in figura 6.1.

Quindi gli utenti possono definire propri task (1), che vengono utilizzati per la definizione di production (2, 3), le quali a loro volta vengono usate per sottomettere al sistema richieste computazionali (4, 5). Le richieste vengono "scomposte" in elementary request (6), che vengono analizzate dal local scheduler (7), responsabile del mapping dei corrispondenti task ai computer disponibili (8).

In questo primo prototipo non si considera quindi il global scheduler: verrà progettato e implementato solo in una seconda fase, in quanto non sono ancora ben definiti i criteri da considerare (ad esempio quando si devono "spostare" i

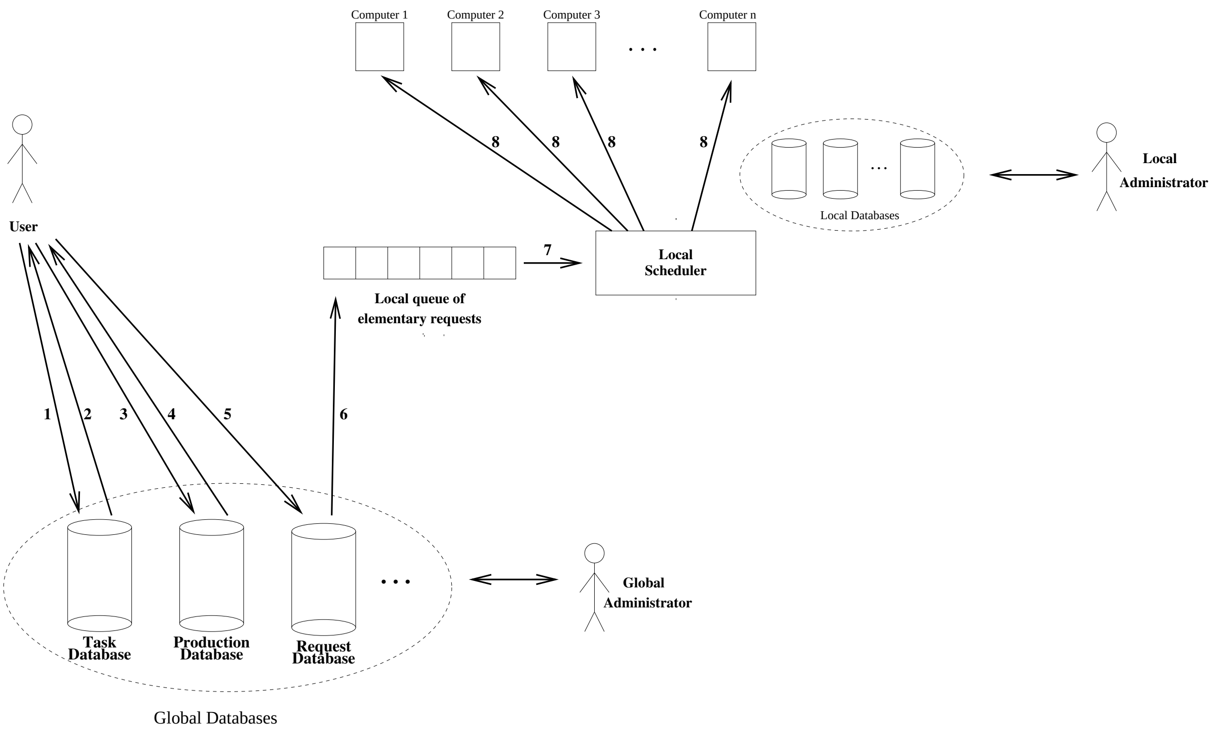


Figura 6.1: Ambiente considerato nella realizzazione del primo prototipo

dati tra i vari siti, o quando si devono invece "spostare" i job che devono essere eseguiti), e non sono ancora del tutto disponibili tutti i necessari strumenti (ad esempio validi tool per il monitor delle linee WAN in tempo reale, in modo da poter valutare l'opportunità dell'uso di tali link per la copia dei dati).

I database globali (Task Database, Production Database, Request Database, Architecture Database, ecc...) vengono gestiti dai global administrator, mentre i database locali (Local Computer Database, Local Buffer Database, Local Tape Device Database, Local Data Database, ecc...) vengono gestiti dai local administrator. Questi database sono comunque tutti fisicamente implementati sullo stesso sito.

6.2 Modello dei dati

Come visto nel capitolo 4, nel sistema DEMI devono essere considerati diversi tipi di informazioni, di natura diversa, fra le quali possono esistere specifiche relazioni, e sulle quali sono possibili determinati tipi di operazioni. È quindi fondamentale poter organizzare tutti questi dati in maniera adeguata, in modo da rendere il più efficiente possibile l'esecuzione di queste operazioni.

Tra i vari modelli esistenti per la gestione delle informazioni, negli ultimi tempi il *modello dei dati a oggetti* sta riscuotendo un grosso interesse, e sono sempre di più le implementazioni di questo modello in sistemi commerciali [6]. Il modello dei dati a oggetti, infatti, si presta bene per specificare in modo naturale la "realtà" che deve essere descritta, in quanto, a differenza degli altri modelli, permette di definire con gli stessi costrutti sia i dati, che le operazioni applicabili su di essi.

Il modello dei dati a oggetti può essere usato per la progettazione concettuale di una base di dati anche se poi si utilizzerà un DBMS non a oggetti (ad esempio un sistema relazionale). Infatti, visto il potere espressivo del modello a oggetti, è in genere conveniente usarlo sempre per la progettazione, per poi trasformare lo schema ottenuto nel modello del sistema considerato [6].

In appendice C vengono introdotti i concetti fondamentali di un modello a oggetti, descrivendo un formalismo grafico, proposto in [6], utilizzato per la definizione degli aspetti fondamentali dello schema di una base di dati.

Nel paragrafo 6.3 viene descritta la progettazione concettuale dello schema della base di dati per il sistema DEMI.

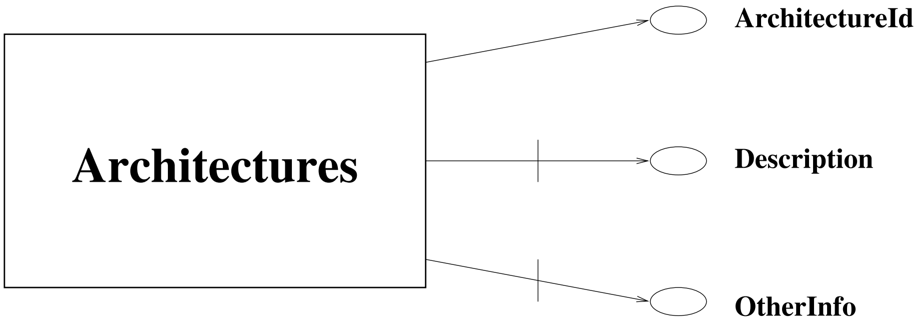


Figura 6.2: Classe 'Architectures'

6.3 Schema della base di dati per il sistema DEMI

In questa sezione viene definito lo schema concettuale della base di dati progettata per la "gestione" delle varie informazioni del sistema DEMI. Vengono prima di tutto elencate le classi utilizzate, specificando per ognuna di esse i campi dei relativi oggetti, ovvero gli attributi utilizzati per la rappresentazione dello stato. Si definiscono poi le associazioni esistenti tra le varie classi.

6.3.1 Architetture

In figura 6.2 è mostrata la classe 'Architectures', che rappresenta le architetture dei calcolatori utilizzati nel sistema DEMI. Quindi ogni elemento di questa classe rappresenta un elemento dell'Architecture Database (vedi par. 4.2.2).
Ad ogni oggetto della classe Architectures sono associati i seguenti campi:

- l'attributo *ArchitectureId* rappresenta l'identificatore dell'architettura (es. *alpha-win32*);
- l'attributo *Description* è la descrizione dell'architettura (es. *Computer con processore Alpha, con sistema operativo Windows NT*);
- l'attributo *OtherInfo* rappresenta eventuali altre informazioni relative all'architettura.

6.3.2 Modelli di tape device

In figura 6.3 è rappresentata la classe 'Tape Device Models', che definisce i modelli di tape device (e dei relativi nastri) utilizzati nel sistema DEMI. Ogni elemento

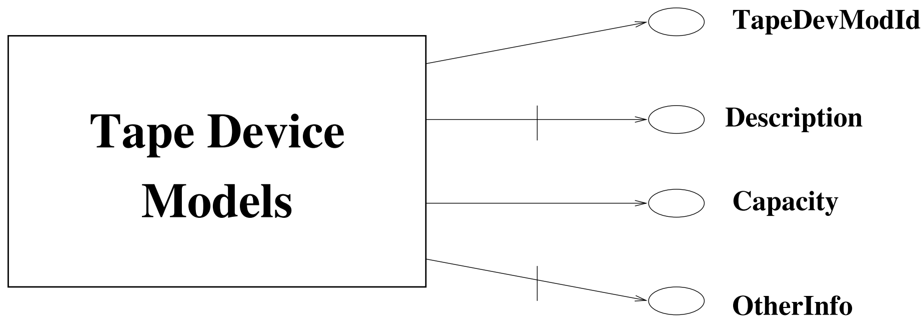


Figura 6.3: Classe 'Tape Device Models'

di questa classe rappresenta un elemento memorizzato nel Tape Device Model Database (vedi par. 4.2.3).
Lo stato di un oggetto di questa classe è definito dai seguenti attributi:

- l'attributo *TapeDevModId*, che rappresenta l'identificatore del modello di tape device (es. *DLT*);
- l'attributo *Description*, che descrive il modello di tape device (es. *tape device DLT, per nastri TZ87*);
- l'attributo *Capacity*, che definisce la capacità di un nastro di questo modello (es. *30 Gbyte*);
- l'attributo *OtherInfo*, che rappresenta eventuali altre informazioni relative a questo modello di tape device.

6.3.3 Global Administrator

In figura 6.4 è definita la classe 'Global Administrators', che rappresenta i global administrator del sistema DEMI. Questa classe corrisponde quindi al Global Administrator Database (vedi par. 4.2.4).
Gli attributi che definiscono lo stato degli oggetti della classe Global Administrators sono:

- *UserId*: specifica lo User Identifier del global administrator (es. *sgaravat*);
- *Password*: rappresenta la password (criptata) del global administrator;
- *EMailAddr*: definisce l'e-mail address del global administrator (es. *sgaravat@domain*);

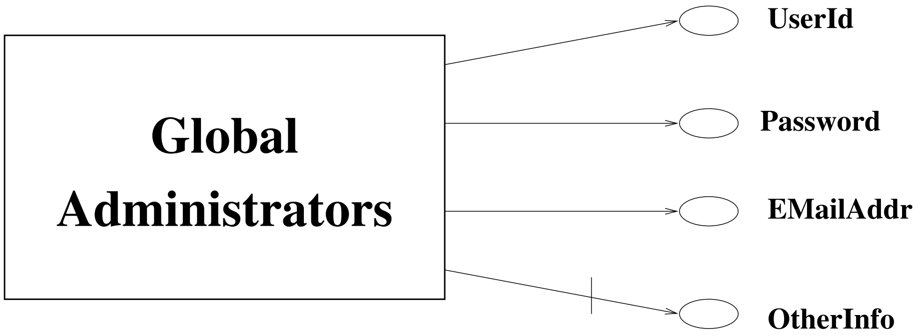


Figura 6.4: Classe 'Global Administrators'

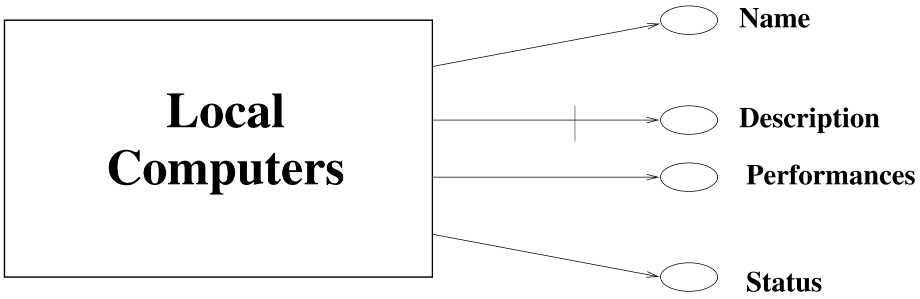


Figura 6.5: Classe 'Local Computers'

- *OtherInfo*: rappresenta eventuali altre informazioni relative al global administrator.

6.3.4 Computer

Per definire i computer presenti nel sito locale, memorizzati nel Local Computer Database (vedi par. 4.3.1), si definisce la classe 'Local Computers', rappresentata in figura 6.5.
Le proprietà degli oggetti di questa classe sono rappresentate da:

- l'attributo *Name*, che definisce il nome o l'indirizzo IP del computer (es. *ntde01.pd.infn.it*);
- l'attributo *Description*, che descrive il computer (es. *alpha station 3000/600, con sistema operativo Windows NT 4.0 Server*);
- l'attributo *Performances*, che indica le prestazioni del computer (è un valore numerico);

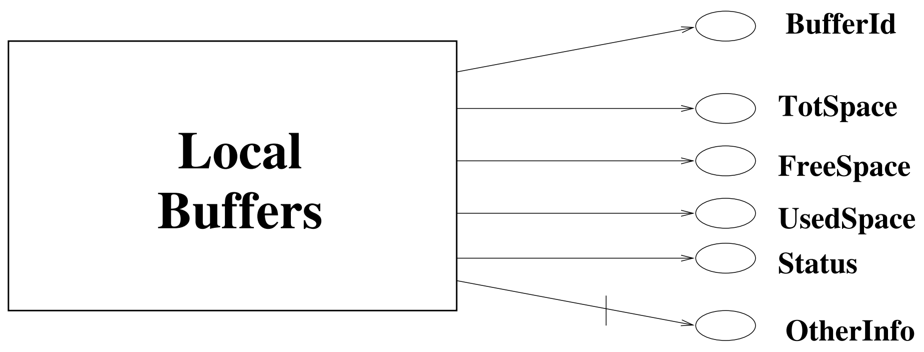


Figura 6.6: Classe 'Local Buffers'

- l'attributo *Status*, che indica se il computer è 'abilitato' o 'disabilitato' (e quindi se può "accettare" o meno job da eseguire).

6.3.5 Buffer

In figura 6.6 è descritta la classe 'Local Buffers', utilizzata per definire i disk buffer presenti nel sito locale, utilizzati dal sistema DEMI per memorizzare i file di dati (quindi i disk buffer definiti nel Local Buffer Database: vedi par. 4.3.2).
Gli attributi che permettono di definire lo stato di un oggetto di tipo Local Buffer sono:

- l'attributo *BufferId*, che rappresenta l'identificatore del disk buffer (es. *disk1*);
- l'attributo *TotSpace*, che definisce lo spazio totale di questo disk buffer assegnato per la memorizzazione di file del sistema DEMI (es. *2 Gbyte*);
- l'attributo *FreeSpace*, che indica lo spazio ancora disponibile in questo buffer (es. *540 Mbyte*);
- l'attributo *UsedSpace*, che indica lo spazio usato nel buffer per la memorizzazione di file (es. *1.3 Gbyte*).

  Il valore *TotSpace* - *FreeSpace* - *UsedSpace* rappresenta lo spazio "allocato" in questo disk buffer (infatti, prima che inizi l'esecuzione di un task, viene allocato lo spazio per la memorizzazione dei suoi output);
- l'attributo *Status*, che indica se il buffer è 'abilitato' o 'disabilitato' (e quindi se può essere utilizzato per la memorizzazione di file);
- l'attributo *OtherInfo*, che rappresenta eventuali altre informazioni relative al buffer.

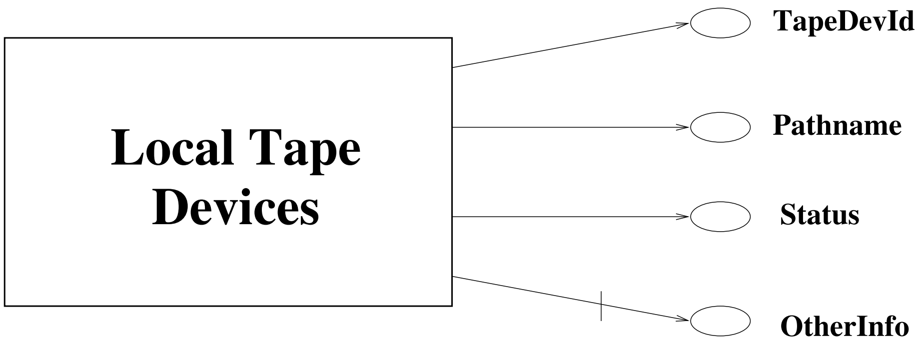


Figura 6.7: Classe 'Local Tape Devices'

6.3.6 Tape device

Per definire gli elementi del Local Tape Device Database, ovvero i tape device presenti nel sito locale, utilizzati dal sistema DEMI (vedi par. 4.3.3), si utilizza la classe 'Local Tape Devices', rappresentata in figura 6.7.
Per definire le proprietà di questi oggetti, si utilizzano i campi TapeDevId, Pathname, Status e OtherInfo.

- L'attributo *TapeDevId* rappresenta l'identificatore del tape device (es. *tapedev1*);
- l'attributo *Pathname* definisce il pathname che permette di "accedere" al tape device (es. */dev/rmt0h*);
- l'attributo *Status* indica se il tape device è 'abilitato' o 'disabilitato' (e quindi se può essere usato per la lettura/scrittura di dati da/su nastro);
- l'attributo *OtherInfo* rappresenta eventuali altre informazioni relative al tape device.

6.3.7 Utenti

In figura 6.8 è rappresentata la classe 'Users', utilizzata per definire gli utenti del sistema DEMI del sito locale, ovvero gli elementi definiti nello User Database (vedi par. 4.3.4).
La classe 'Users' è "costituita" dalle sottoclassi 'Local Administrators', 'Operators' e 'General Users'; si tratta di sottoclassi copertura, in quanto l'unione dei loro elementi coincide con gli elementi della classe 'Users', ma questi insiemi in generale non sono tra loro disgiunti: un utente può infatti avere più "ruoli" (ad esempio può essere allo stesso tempo un general user e un operator).

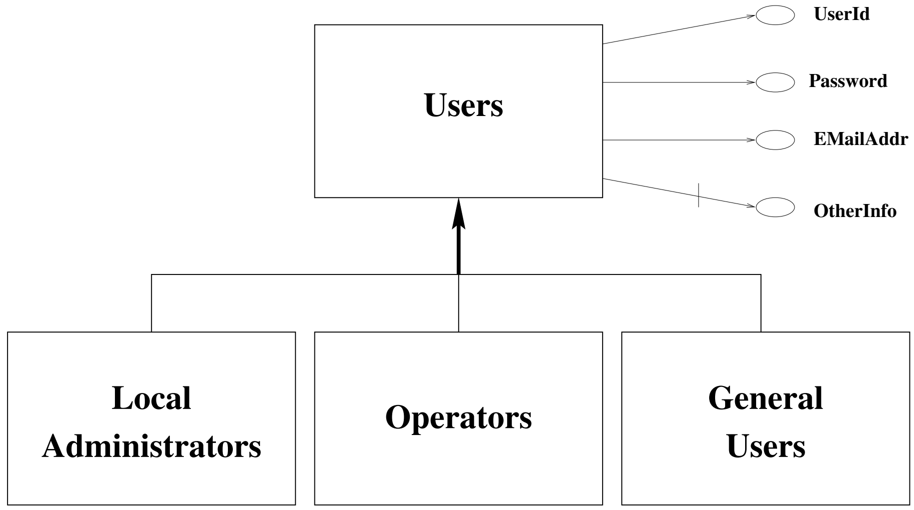


Figura 6.8: Classi 'Users', 'Local Administrators', 'Operators', 'General Users'

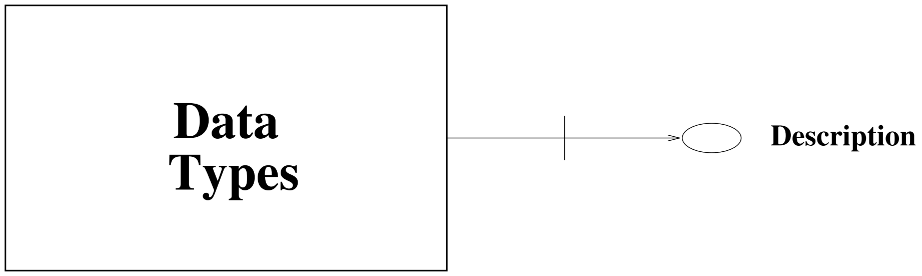


Figura 6.9: Classe 'Data Types'

- L'attributo *UserId* rappresenta lo User Identifier dell'utente (es. *rossi*);
- l'attributo *Password* rappresenta la password (criptata) dell'utente;
- l'attributo *EMailAddr* definisce l'e-mail address dell'utente (es. *rossi@local-domain*);
- l'attributo *OtherInfo* definisce eventuali altre informazioni relative all'utente.

6.3.8 Dati

In figura 6.9 è rappresentata la classe 'Data Types', che definisce i possibili tipi associati ai file di dati.

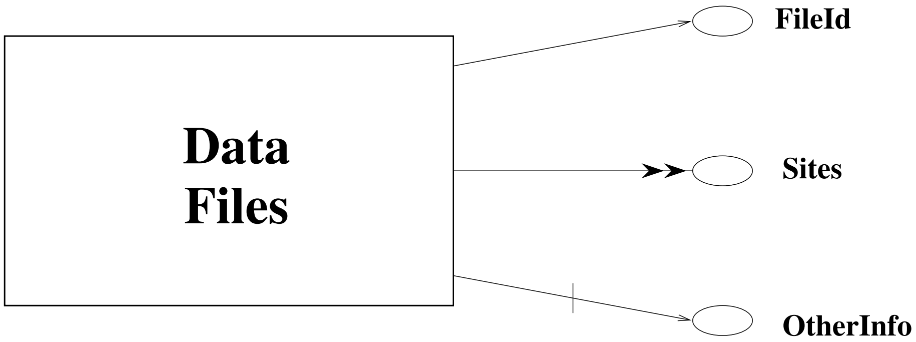


Figura 6.10: Classe 'Data Files'

Per definire lo stato di un oggetto di questa classe è sufficiente utilizzare l'attributo *Description*, che rappresenta la descrizione del tipo di dati (es. *QQPS, 183 GeV, 97_1 MC*).

I file di dati definiti nel Global Data Database (vedi par. 4.4.1) vengono rappresentati attraverso la classe 'Data Files', mostrata in figura 6.10.
I campi di un oggetto di questa classe sono:

- *FileId*: rappresenta l'identificatore del file di dati (es. *fileid1*);
- *Sites*: indica in che siti il file di dati è fisicamente presente: si tratta di un attributo multivalore, in quanto uno stesso file può essere presente in più siti. Poiché, come detto, in questa prima implementazione si considera un sistema composto da un solo sito, per il momento questo attributo non è significativo;
- *OtherInfo*: rappresenta eventuali altre informazioni relative al file di dati.

In figura 6.11 è rappresentata la classe 'Local Data Files', che definisce gli elementi memorizzati nel Local Data Database, ovvero i file di dati presenti nel sito locale considerato (vedi par. 4.4.2).
Lo stato di un oggetto di questo tipo viene definito dai seguenti attributi:

- l'attributo *FileId*, che rappresenta l'identificatore del file di dati (es. *fileid1*);
- l'attributo *Size*, che definisce la dimensione del file (es. *257 Mbyte*);
- l'attributo *OtherInfo*, che definisce eventuali altre informazioni.

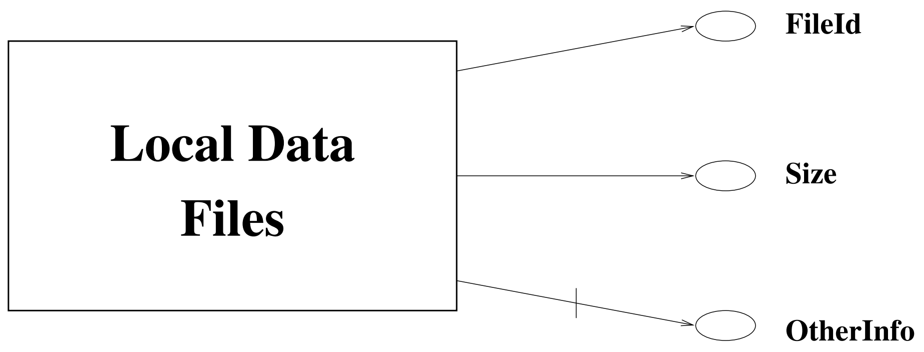


Figura 6.11: Classe 'Local Data Files'

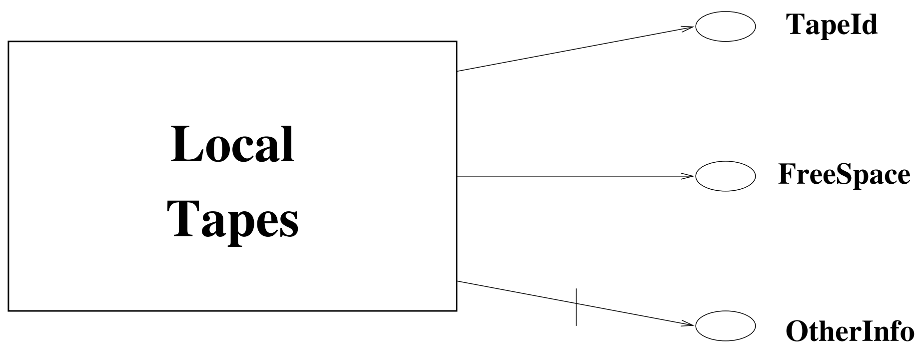


Figura 6.12: Classe 'Local Tapes'

La classe 'Local Tapes', rappresentata in figura 6.12, viene utilizzata per definire gli elementi del Local Tape Database (vedi par. 4.4.3), ovvero i nastri presenti nel sito locale, utilizzati per memorizzare file di dati.
Le proprietà di questi oggetti sono definiti da:

- l'attributo *TapeId*, che rappresenta l'identificatore del nastro (es. *tape1*);
- l'attributo *FreeSpace*, che indica quanto spazio è ancora disponibile sul nastro (es. *8 Gbyte*);
- l'attributo *OtherInfo*, che definisce eventuali altre informazioni.

6.3.9 Task

In figura 6.13 è rappresentata la classe 'Task Types', che definisce i tipi di task: ogni elemento di questa classe corrisponde ad un elemento del Task Database (vedi par. 4.5.1).

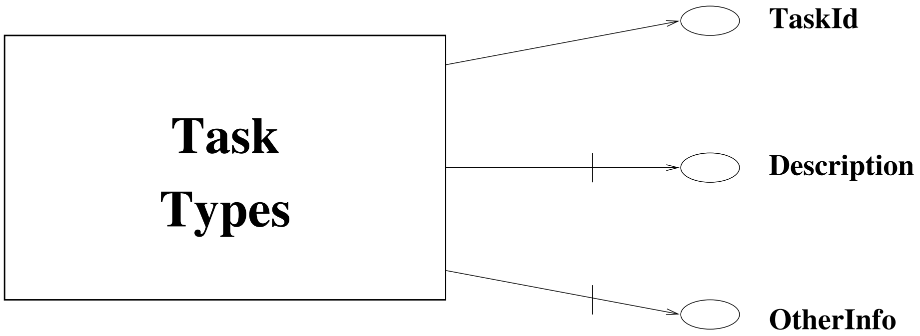


Figura 6.13: Classe 'Task Types'

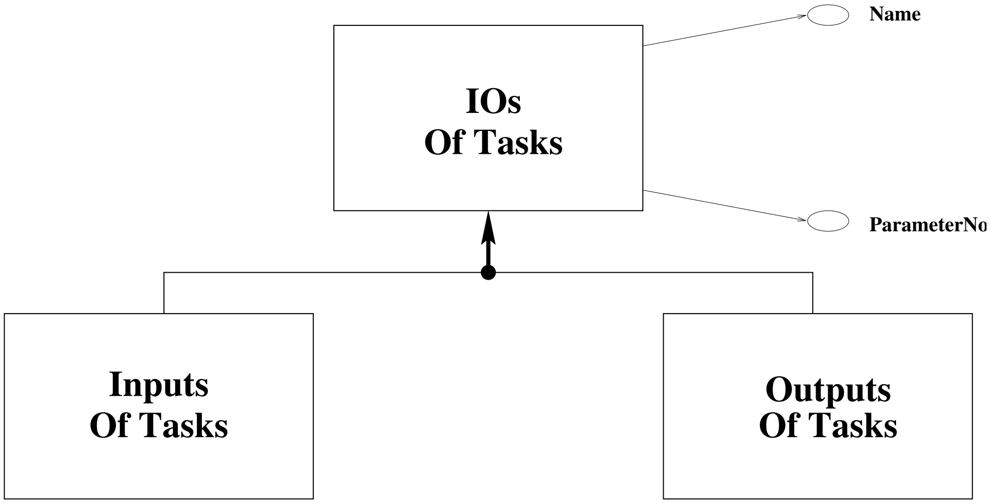


Figura 6.14: Classi 'IOs of tasks', 'Inputs Of Tasks', 'Outputs Of Tasks'

Gli attributi che vengono usati per definire lo stato di un oggetto di questa classe sono:

- *TaskId*, che rappresenta l'identificatore del tipo di task (es. *TaskTypeX*);
- *Description*, che descrive il tipo di task (es. *Task che gestisce l'attività X*);
- *OtherInfo*, che definisce eventuali altre informazioni.

In figura 6.14 è rappresentata la classe 'IOs Of Tasks', che definisce gli input-output dei tipi di task.

La classe 'IOs Of Tasks' è "costituita" dalle sottoclassi 'Inputs Of Tasks' e 'Outputs Of Tasks' (si tratta di una partizione).
Le proprietà di un oggetto di tipo 'IO Of Tasks' sono definite da:

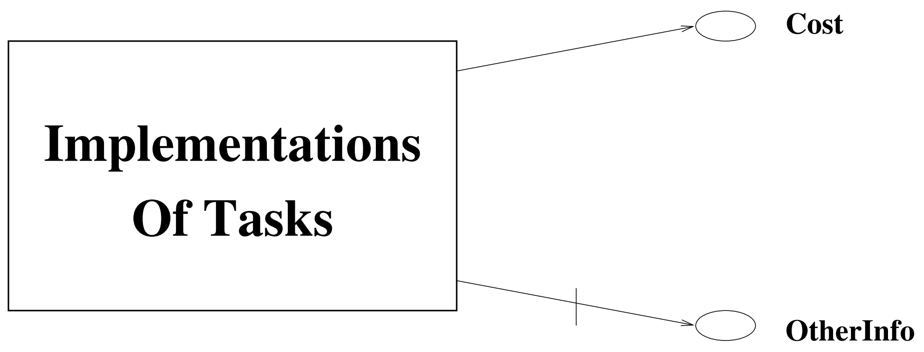


Figura 6.15: Classe 'Implementations Of Tasks'

- l'attributo *Name*, che rappresenta il nome dell'input/output considerato;
- l'attributo *ParameterNo*, che definisce il numero ordinale di parametro con cui si identifica questo input/output nella riga di comando che "invoca" l'esecuzione del corrispondente task. Infatti, come visto nel paragrafo 4.5.1, è necessario definire una corrispondenza tra input-output dei task, e i parametri che devono essere specificati nelle corrispondenti linee di comando.

L'implementazione di un task per una specifica architettura di calcolatori viene definita attraverso la classe 'Implementations Of Tasks', rappresentata in figura 6.15.
Gli attributi necessari per specificare le proprietà di questi oggetti sono:

- *Cost*: rappresenta il parametro 'Cost of the task';
- *OtherInfo*: definisce eventuali altre informazioni relative a questa implementazione.

In figura 6.16 è rappresentata la classe 'Files For Implementations', usata per definire i file necessari per l'esecuzione dei tipi di task, per le sue varie implementazioni considerate.

La classe 'Files For Implementations' ha come sottoclasse la classe 'Executable Files', usata per definire il file che deve essere indicato nella riga di comando specificata per l'esecuzione (vedi par. 4.5.1).

In generale per un'implementazione di un tipo di task per una certa architettura sono necessari vari file, uno dei quali, definito come 'eseguibile', viene specificato nella riga di comando che si utilizza quando si deve invocare l'esecuzione.

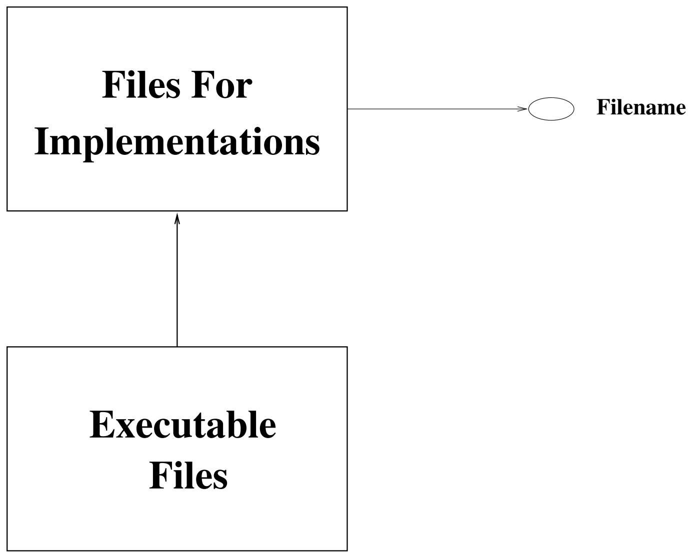


Figura 6.16: Classi 'Files For Implementations' e 'Executable Files'

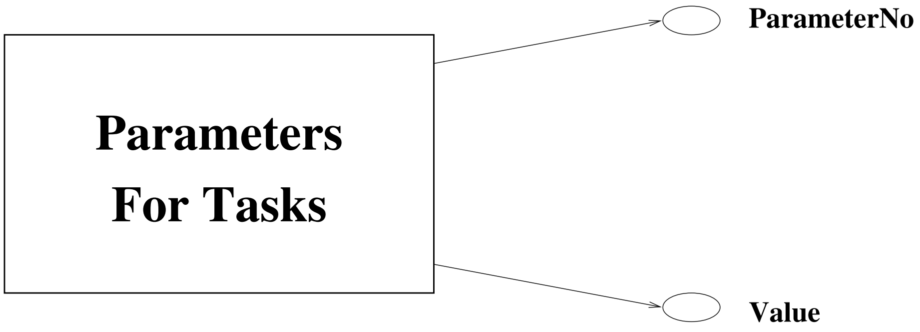


Figura 6.17: Classe 'Parameters For Tasks'

L'attributo *FileName* rappresenta il pathname del file.

Oltre ai parametri che definiscono gli input e gli output, quando si deve eseguire un task può essere necessario specificare nella corrispondente riga di comando altri parametri. Questi vengono definiti attraverso la classe 'Parameters For Tasks', rappresentata in figura 6.17.
Le proprietà degli oggetti di questo tipo sono definite dagli attributi:

- *ParameterNo*: rappresenta il numero ordinale del parametro (primo, secondo, ecc...);
- *Value*: specifica il valore associato al parametro.

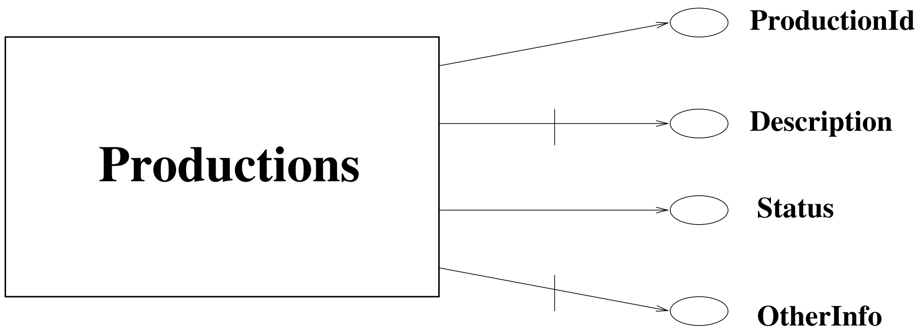


Figura 6.18: Classe 'Productions'

6.3.10 Production

In figura 6.18 è rappresentata la classe 'Productions', utilizzata per definire le produzioni; ogni oggetto di questa classe corrisponde ad un elemento del Production Database (vedi par. 4.5.2).
Per descrivere lo stato di un oggetto di tipo Production, si utilizzano gli attributi:

- *ProductionId*, che rappresenta l'identificatore della production (es. *ProdX*);
- *Description*, che descrive la produzione (es. *Production che gestisce l'elaborazione Y*);
- *Status*, che indica se la production può essere usata o meno per sottomettere nuove computational request;
- *OtherInfo*, che rappresenta eventuali altre informazioni relative alla produzione considerata.

Una produzione è costituita da più task: questi vengono rappresentati attraverso la classe 'Tasks Of Productions', descritta in figura 6.19.
L'attributo *Name* rappresenta il nome del task.

È necessario definire anche i data box delle produzioni. A tal fine si utilizza la classe 'Data Boxes Of Productions', rappresentata in figura 6.20.
La classe 'Data Boxes Of Productions' è "costituita" dalle sottoclassi 'Inputs Of Productions', 'Temporary Data Of Productions' e 'Outputs of productions' (si tratta di una partizione).
La classe 'Inputs of productions' è "costituita" dalle sottoclassi 'Singleton Inputs Of Productions' e 'Stream Inputs Of Productions' (una partizione).

Figura 6.19: Classe 'Tasks Of Productions'

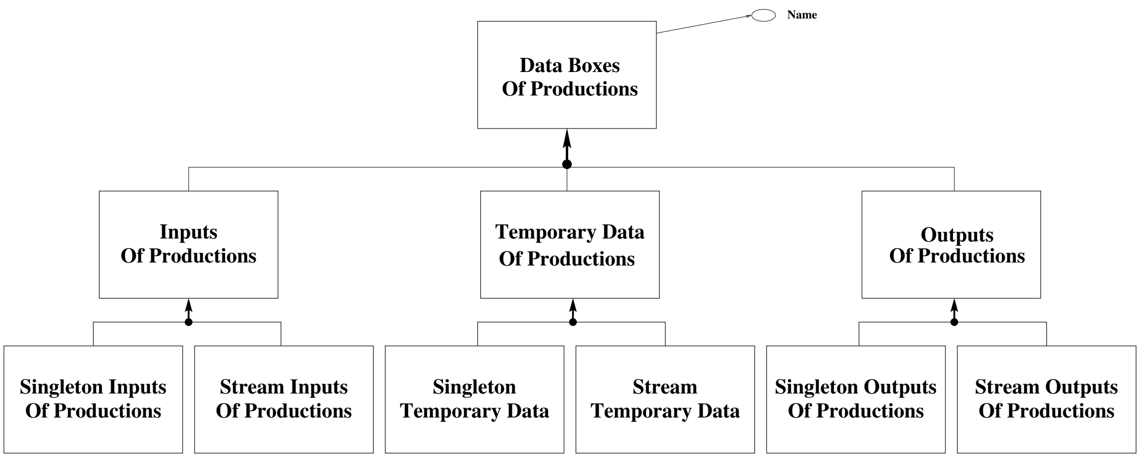


Figura 6.20: Classi 'Data Boxes Of Productions', 'Inputs Of Productions', 'Temporary Data Of Productions', 'Outputs Of Productions', 'Singleton Inputs Of Productions', 'Stream Inputs Of Productions', 'Singleton Temporary Data', 'Stream Temporary Data', 'Singleton Outputs Of Productions', 'Stream Outputs Of Productions'

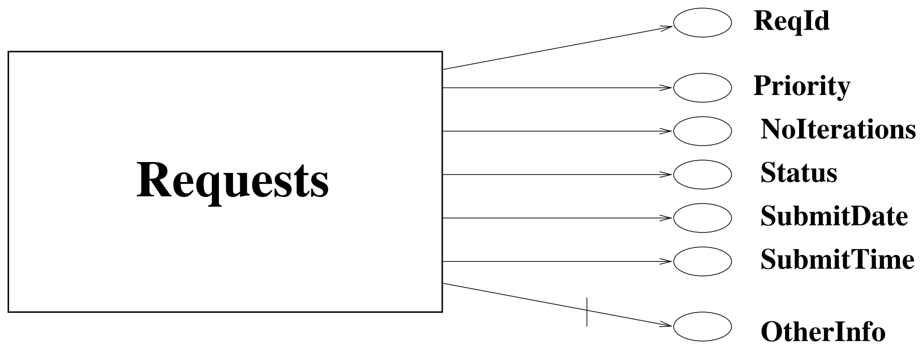


Figura 6.21: Classe 'Requests'

Una partizione analoga viene definita anche per le classi 'Temporary Data Of Productions' e 'Outputs Of Productions'.

I data box delle produzioni vengono quindi "divisi" in base al loro ruolo (input, output, temporanei), e poi vengono classificati in base al loro tipo (stream o singleton).

Il campo *Name* rappresenta il nome del data box.

6.3.11 Computational request

In figura 6.21 è rappresentata la classe 'Requests', usata per definire le computational request sottomesse al sistema DEMI dagli utenti. Ogni oggetto di questa classe corrisponde ad un elemento del Request Database (vedi par. 4.5.3).
Lo stato degli oggetti di tipo Request è definito dagli attributi:

- *ReqId*, che rappresenta l'identificatore della computational request;
- *Priority*, che indica il livello di priorità;
- *NoIterations*, che indica il numero di volte in cui deve essere eseguita la relativa production (nel caso in cui non ci siano input di tipo stream);
- *Status*, che indica lo stato di elaborazione della richiesta (soddisfatta, processing in corso, ecc...);
- *SubmitDate*, che rappresenta la data in cui la richiesta è stata sottomessa al sistema da parte di un utente;
- *SubmitTime*, che rappresenta l'ora in cui la richiesta è stata sottomessa al sistema DEMI;

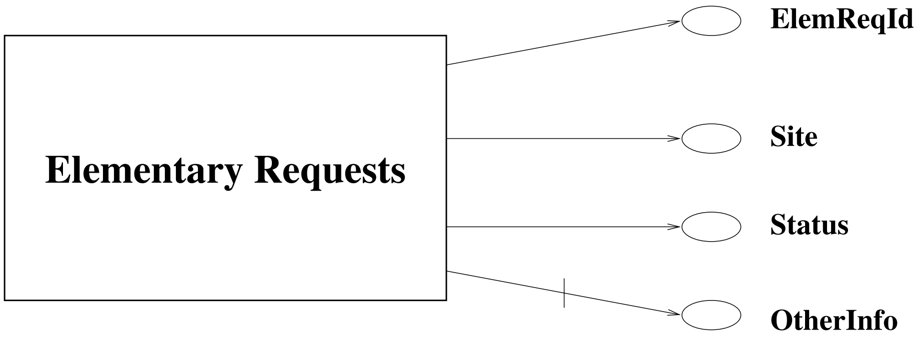


Figura 6.22: Classe 'Elementary requests'

- *OtherInfo*, che definisce eventuali altre informazioni.

6.3.12 Processing

Una computational request viene "scomposta" in elementary request, ognuna delle quali deve poi essere elaborata. Per la definizione di queste entità si utilizza la classe 'Elementary Requests', rappresentata in figura 6.22. Ogni elemento di questa classe corrisponde ad un elemento del Global Processing Database (vedi par. 4.8.1). Gli attributi usati per definire lo stato di questi oggetti sono:

- *ElemReqId*: rappresenta l'elementary request identifier (l'identificatore dell'elementary request);
- *Site*: indica in che sito l'elementary request è stata assegnata per la relativa elaborazione (considerando un ambiente costituito da un unico sito, questo attributo per il momento non è significativo);
- *Status*: indica lo stato dell'elaborazione di questa elementary request (in esecuzione, elaborazione completata con successo, processing fallito, ecc...);
- *OtherInfo*: definisce eventuali altre informazioni.

In figura 6.23 è rappresentata la classe 'Local Elementary Processings', utilizzata per definire le elaborazioni delle elementary request "assegnate" al sito locale (ovvero gli elementi del Local Processing Database: vedi par. 4.8.2).
Gli attributi da considerare sono:

- *ElemReqId*, che rappresenta l'elementary request identifier;

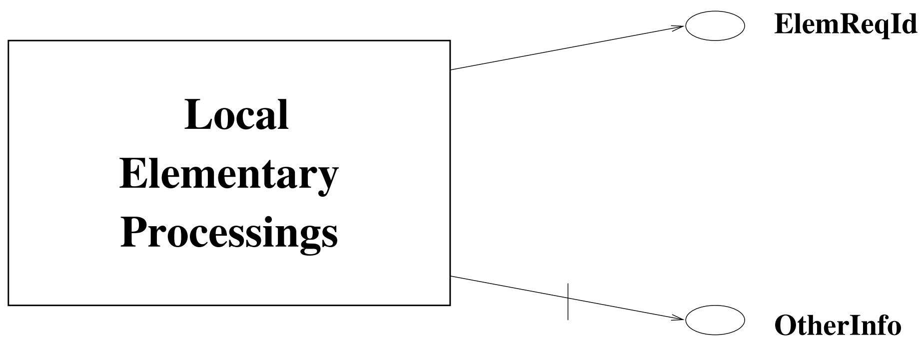


Figura 6.23: Classe 'Local Elementary Processings'

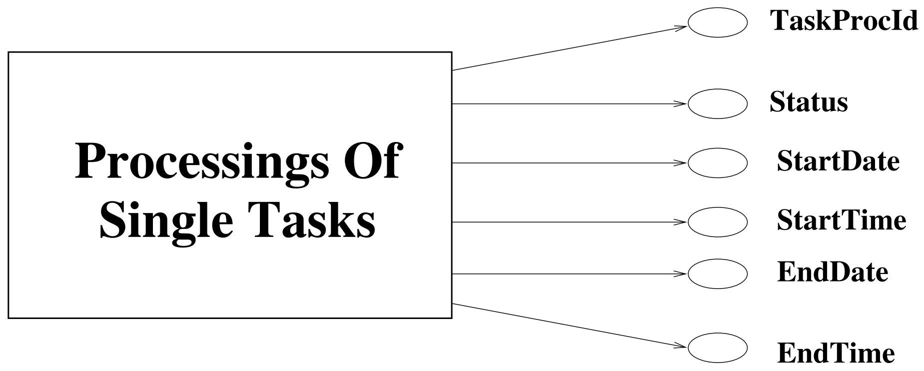


Figura 6.24: Classe 'Processings Of Single tasks'

- *OtherInfo*, che definisce eventuali altre informazioni.

Nel Processing Database è necessario "descrivere" anche le elaborazioni dei singoli task componenti la production, per ogni elementary request considerata. A tal fine si considera la classe 'Processings Of Single Tasks', rappresentata in figura 6.24. Gli attributi da considerare per la descrizione dello stato sono:

- l'attributo *TaskProcId*: rappresenta l'identificatore del processing;
- l'attributo *Status*: definisce lo stato dell'elaborazione del task (in esecuzione, elaborazione completata con successo, processing fallito, ecc...);
- l'attributo *StartDate*: indica la data in cui è iniziata l'esecuzione del task;
- l'attributo *StartTime*: indica l'ora in cui è iniziata l'elaborazione del task;
- l'attributo *EndDate*: indica la data in cui è terminata l'esecuzione del task;
- l'attributo *EndTime*: indica l'ora in cui è terminata l'elaborazione del task.

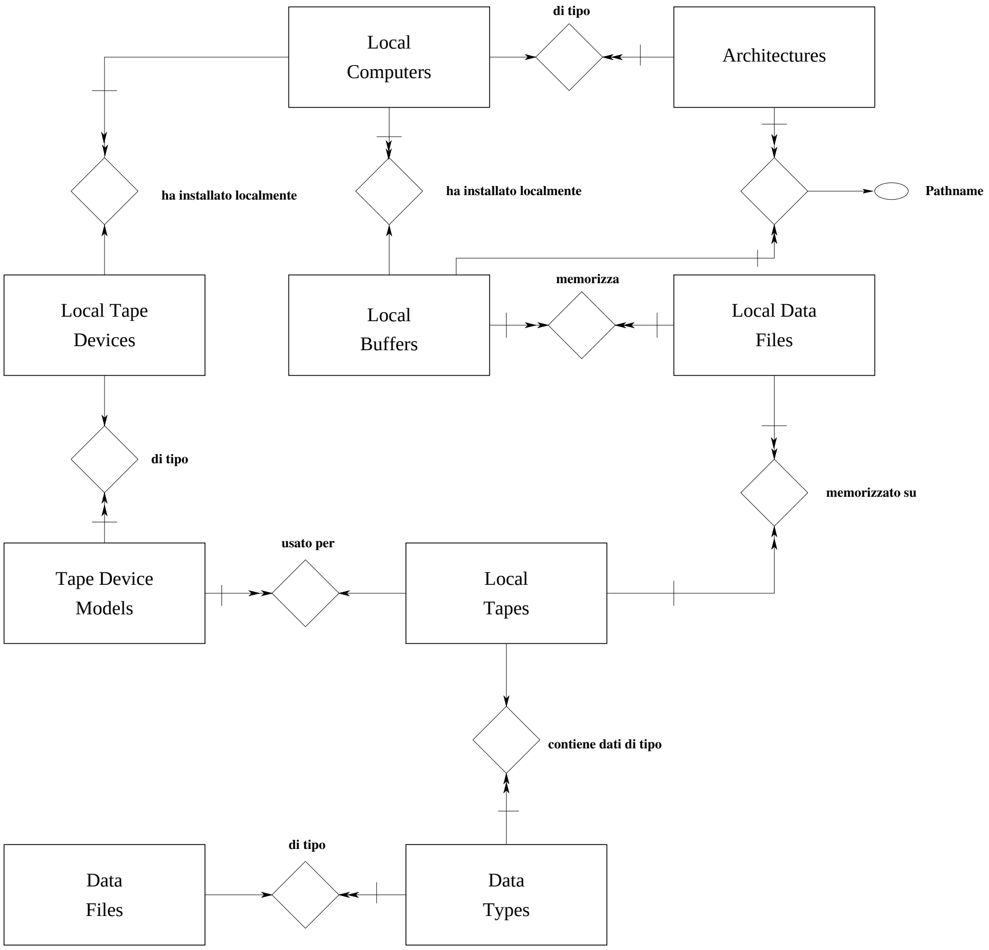


Figura 6.25: Schema della base di dati (parte I)

6.3.13 Associazioni

In questo paragrafo vengono definite le associazioni esistenti tra le varie classi prima definite.

In figura 6.25 sono rappresentate le associazioni tra le classi 'Local Computers', 'Architectures', 'Local Tape Devices', 'Local Buffers', 'Local Data Files', 'Tape Device Models', 'Local Tapes', 'Data Types' e 'Data Files'.

- Associazione tra 'Local Computers' e 'Architectures'.

  Si definisce che ogni computer è di una specifica architettura.

  La relazione diretta è univoca e totale (un computer è di un'unica architettura). La relazione inversa è multivalore (possono esserci più computer di una certa architettura) e parziale (potrebbero non esserci computer di una certa architettura).

- Associazione tra 'Local Computers' e 'Local Buffers'.

  Si specifica che i disk buffer sono fisicamente “installati” su computer.

  La relazione diretta è multivalore (su un computer possono essere installati più disk buffer) e parziale (su un computer potrebbero non essere presenti buffer). La relazione inversa è univoca e totale (un disk buffer è fisicamente installato su un unico computer).

- Associazione tra 'Local Buffers' e 'Architectures'.

  Questa associazione con attributo definisce che ad ogni buffer è associato un pathname, per ogni architettura (quindi tutti i computer della stessa architettura accedono al buffer con uno stesso pathname[1]).

- Associazione tra 'Local Computers' e 'Local Tape Devices'.

  La relazione indica che sui computer possono essere installati tape device.

  La relazione diretta è multivalore (su un computer possono essere installati più tape device) e parziale (su un computer potrebbero non essere installati tape device). La relazione inversa è univoca e totale (un tape device è fisicamente installato su un unico computer).

- Associazione tra 'Local Tape Devices' e 'Tape Device Models'.

  Si definisce che i tape device sono di specifici tipi.

  La relazione diretta è univoca e totale (un tape device è di un unico modello). La relazione inversa è multivalore (possono esserci più tape device di un certo modello) e parziale (potrebbero non esserci tape device di un certo modello).

- Associazione tra 'Local Buffers' e 'Local Data Files'.

  Si indica che i disk buffer vengono usati per memorizzare file di dati.

  La relazione diretta è multivalore (su un disk buffer possono essere memorizzati più file) e parziale (un disk buffer potrebbe essere “vuoto”, senza file memorizzati). La relazione inversa è multivalore (uno stesso file può essere memorizzato su più disk buffer) e parziale (un file potrebbe non essere memorizzato in nessun disk buffer: potrebbe essere presente solo su nastro).

[1] Sarebbe stato più corretto definire questa associazione tra le classi 'Local Buffers' e 'Local Computers', in quanto non necessariamente due computer della stessa architettura accedono ad uno stesso buffer con lo stesso pathname. In questo primo prototipo si è scelto questo approccio per semplificare la struttura.

- Associazione tra 'Local Data Files' e 'Local Tapes'.

  Si indica che i nastri vengono usati per memorizzare file di dati.

  La relazione diretta è multivalore (uno stesso file può essere memorizzato su più nastri) e parziale (un file potrebbe non essere memorizzato in nessun nastro: potrebbe essere presente solo su disco). La relazione inversa è multivalore (su un nastro possono essere memorizzati più file) e parziale (un nastro potrebbe essere "vuoto", senza file memorizzati).

- Associazione tra 'Local Tapes' e 'Tape Device Models'.

  La relazione specifica che ogni nastro è di un certo tipo (i modelli dei tape device corrispondono anche ai modelli dei relativi nastri).

  La relazione diretta è univoca e totale (un nastro è di un unico modello). La relazione inversa è multivalore (possono esserci più nastri di un certo modello) e parziale (potrebbero non esserci nastri di un certo modello).

- Associazione tra 'Local Tapes' e 'Data Types'.

  Si indica che su un nastro vengono memorizzati file di dati tutti di uno stesso tipo.

  La relazione diretta è univoca e totale (un nastro contiene dati di un unico tipo). La relazione inversa è multivalore (possono esserci più nastri che contengono dati di un certo tipo) e parziale (potrebbero non esserci nastri che contengono file di un certo tipo).

- Associazione tra 'Data Files' e 'Data Types'.

  Si definisce che ogni file di dati è di un certo tipo.

  La relazione diretta è univoca e totale (un file di dati è di un unico tipo). La relazione inversa è multivalore (possono esserci più file di un certo tipo) e parziale (potrebbero non esserci file di un certo tipo).

In figura 6.26 sono rappresentate le associazioni tra le classi 'General Users', 'Task Types', 'Inputs Of Tasks', 'Outputs Of Tasks', 'Architectures', 'Implementations Of Tasks', 'Parameters For Tasks', 'Files For Implementations' e 'Executable Files'.

- Associazione tra 'General Users' e 'Task Types'.

  Si indica che i general user possono definire tipi di task.

  La relazione diretta è multivalore (un general user può aver definito più tipi di task) e parziale (un general user potrebbe non aver definito nessun tipo

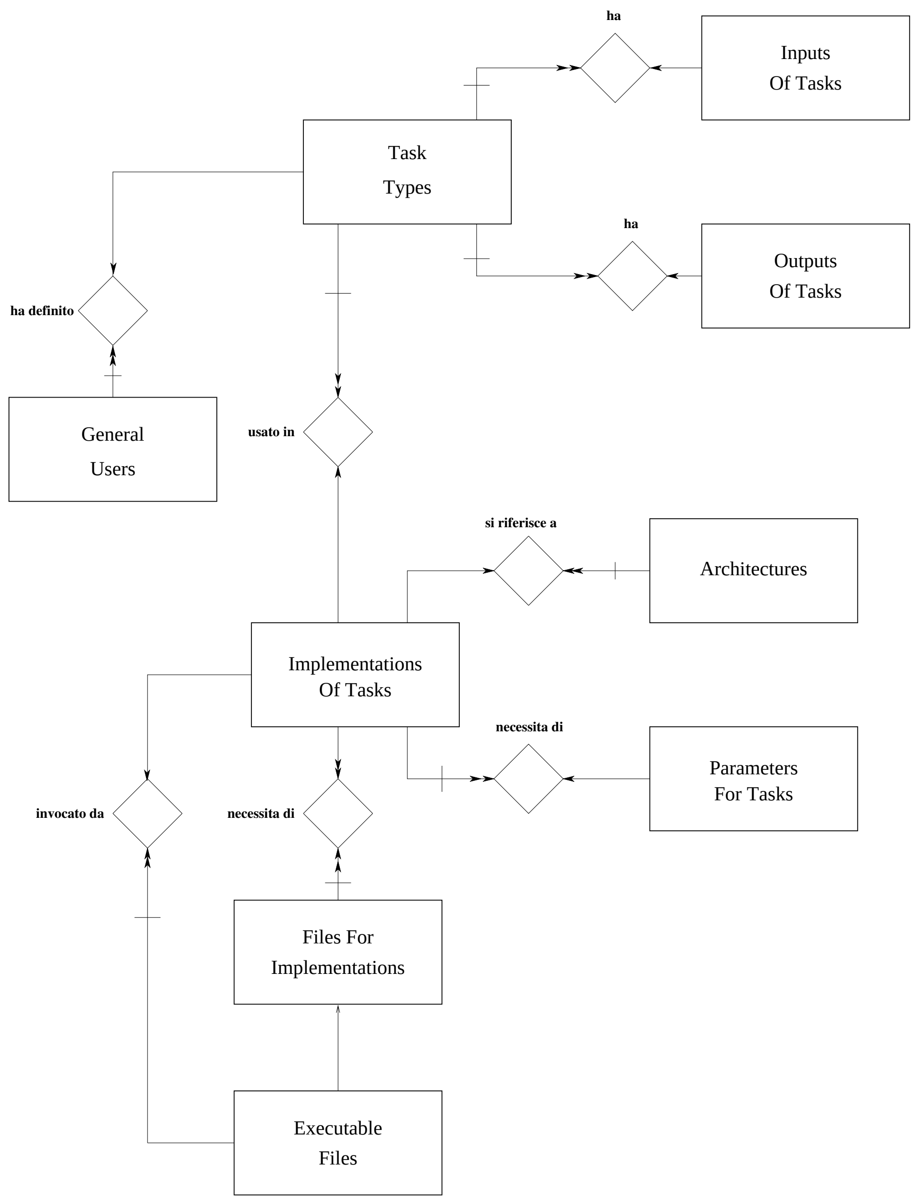


Figura 6.26: Schema della base di dati (parte II)

di task). La relazione inversa è univoca (un tipo di task è stato definito da un unico general user) e totale (un tipo di task ha sempre un utente che l'ha definito).

- Associazione tra 'Task Types' e 'Inputs Of Tasks'.

  Si specifica che un tipo di task può avere input.

  La relazione diretta è multivalore (un tipo di task può avere più input) e parziale (un tipo di task potrebbe non avere nessun input). La relazione inversa è univoca e totale (uno specifico input è associato ad un unico tipo di task).

- Associazione tra 'Task Types' e 'Outputs Of Tasks'.

  Si definisce che un tipo di task può avere output.

  La relazione diretta è multivalore (un tipo di task può avere più output) e parziale (un tipo di task potrebbe non avere output). La relazione inversa è univoca e totale (uno specifico output è associato ad un unico tipo di task).

- Associazione tra 'Task Types' e 'Implementations Of Tasks'.

  La relazione indica che i tipi di task possono essere implementati per diverse architetture.

  La relazione diretta è multivalore (ad un tipo di task possono essere associate più implementazioni) e parziale (un tipo di task potrebbe non essere stato implementato per nessuna architettura). La relazione inversa è univoca e totale (un'implementazione è associata ad un unico tipo di task).

- Associazione tra 'Implementations Of Tasks' e 'Architectures'.

  Si specifica che un'implementazione di un tipo di task si riferisce ad una specifica architettura di calcolatori.

  La relazione diretta è univoca e totale (ad una implementazione è associata un'unica architettura). La relazione inversa è multivalore (un'architettura potrebbe essere usata in più implementazioni) e parziale (un'architettura potrebbe non essere usata in nessuna implementazione).

- Associazione tra 'Implementations Of Tasks' e 'Files For Implementations'.

  La relazione rappresenta il fatto che, per eseguire un tipo di task, considerando una sua specifica implementazione per un'architettura, sono indispensabili dei file.

La relazione diretta è multivalore (per una implementazione possono essere necessari più file) e totale (per una implementazione è sempre necessario almeno un file). La relazione inversa è multivalore (uno stesso file potrebbe essere usato in diverse implementazioni) e parziale (un file potrebbe non essere usato in nessuna implementazione).

- Associazione tra 'Implementations Of Tasks' e 'Executable files'.

  Si specifica che per eseguire un task, considerando una sua specifica implementazione, deve essere invocato uno specifico programma eseguibile/script.

  La relazione diretta è univoca e totale (ad una implementazione è associato un unico file eseguibile). La relazione inversa è multivalore (un file eseguibile/script potrebbe essere usato in più implementazioni) e parziale (un file eseguibile potrebbe non essere usato in nessuna implementazione).

- Associazione tra 'Implementations Of Tasks' e 'Parameters For Tasks'.

  La relazione indica che per eseguire un task, considerando una sua specifica implementazione, può essere necessario specificare nella riga di comando alcuni parametri (oltre a quelli di input - output).

  La relazione diretta è multivalore (ad una implementazione di un tipo di task possono essere associati più parametri) e parziale (per una implementazione di un task potrebbe non essere necessario specificare parametri nella riga di comando, oltre ai parametri di input-output). La relazione inversa è univoca e totale (uno specifico parametro è associato ad un unica implementazione).

In figura 6.27 sono rappresentate le associazioni tra le classi 'General Users', 'Productions', 'Tasks Of Productions', 'Data Boxes Of Productions', 'IOs Of Tasks', 'Task Types'.

- Associazione tra 'General Users' e 'Productions'.

  Si indica che i general user possono definire produzioni.

  La relazione diretta è multivalore (un general user può aver definito più production) e parziale (un general user potrebbe non aver definito nessuna production). La relazione inversa è univoca (una production è definita da un unico general user) e totale (una production ha sempre un utente che l'ha definita).

- Associazione tra 'Productions' e 'Tasks Of Productions'.

  Si indica che le production sono costituite da task.

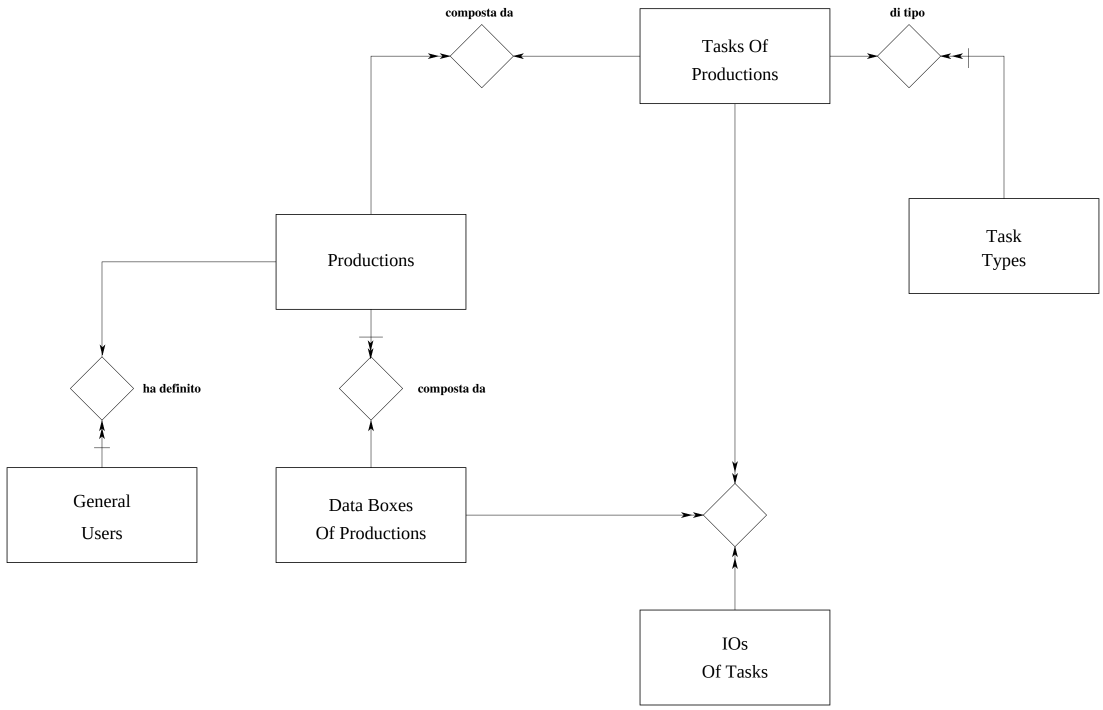


Figura 6.27: Schema della base di dati (parte III)

La relazione diretta è multivalore (una production può essere composta da più task) e totale (una production è composta da almeno un task). La relazione inversa è univoca e totale (uno specifico task di una production è associato ad un'unica production).

- Associazione tra 'Tasks Of Productions' e 'Task Types'.

  La relazione specifica che ogni task componente di una produzione è di un certo tipo.

  La relazione diretta è univoca e totale (un task è di un unico tipo). La relazione inversa è multivalore (ad un tipo di task possono essere associati più task componenti di production) e parziale (un tipo di task potrebbe non essere usato in nessun task componente di production).

- Associazione tra 'Productions' e 'Data Boxes Of Productions'.

  Si indica che nelle production si devono considerare i data box.

  La relazione diretta è multivalore (una production può essere composta da più data box) e parziale (in una production potrebbero non esserci data box). La relazione inversa è univoca e totale (uno specifico data box di una production è associato ad una singola production).

- Associazione tra 'Data Boxes Of Productions', 'Tasks Of Productions', 'IOs Of Tasks'.

  Questa associazione ternaria viene utilizzata per definire le relazioni tra i task che compongono le produzioni. Si definisce che ad ogni data box di una produzione corrispondono uno o più input/output dei task che la compongono.

  Come descritto in [6], per l'effettiva implementazione di questa associazione si può definire una quarta classe, e definire associazioni tra questa classe e, rispettivamente, le classi 'Data Boxes Of Productions', 'Tasks Of Productions' e 'IOs Of Tasks'.

In figura 6.28 sono rappresentate le associazioni tra le classi 'General Users', 'Requests', 'Productions', 'Data Boxes Of Productions', 'Outputs Of Productions', e 'Data Types'.

- Associazione tra 'General Users' e 'Requests'.

  Si indica che i general user possono sottomettere computational request.

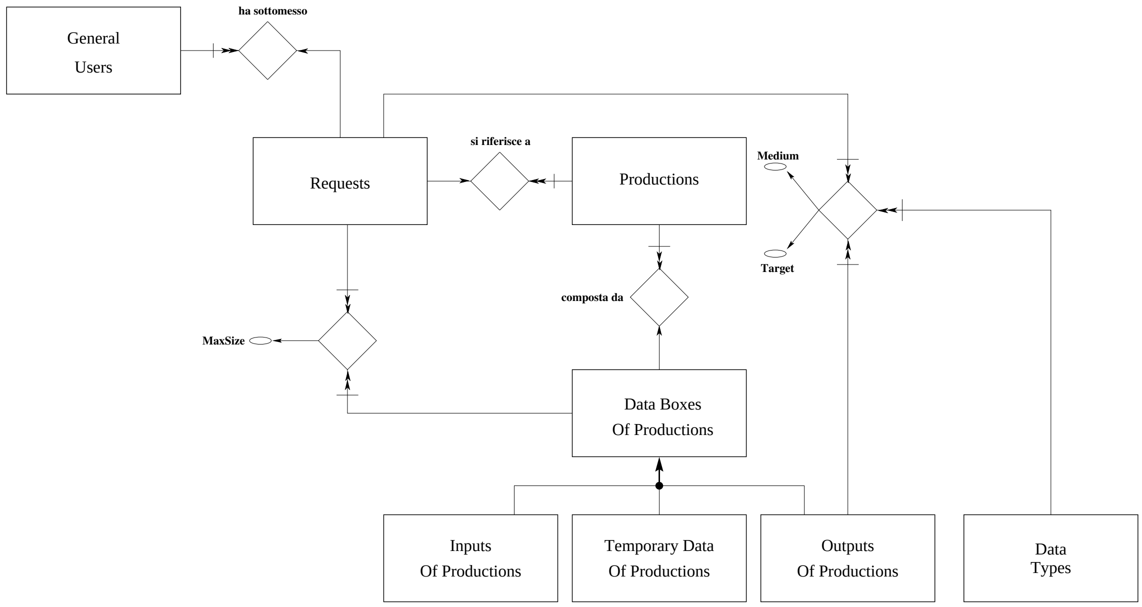


Figura 6.28: Schema della base di dati (parte IV)

La relazione diretta è multivalore (un general user può aver sottomesso più richieste) e parziale (un general user potrebbe non avere sottomesso nessuna richiesta). La relazione inversa è univoca e totale (una richiesta è sottomessa da un unico general user).

- Associazione tra 'Requests' e 'Productions'.

  La relazione indica che ogni richiesta fa riferimento ad una specifica produzione.

  La relazione diretta è univoca e totale (una computational request fa riferimento ad un'unica production). La relazione inversa è multivalore (una produzione può essere usata in più richieste) e parziale (una produzione potrebbe non essere stata usata in nessuna richiesta).

- Associazione tra 'Requests' e 'Data Boxes Of Productions'.

  Questa associazione con attributo indica che, per ogni richiesta, si devono "descrivere" i data box della corrispondente produzione (per definire la massima dimensione di un file associato al data box considerato, prodotto in questa richiesta).

  Per l'effettiva implementazione si potrà definire una nuova classe, con un attributo 'MaxSize', con associazioni tra questa classe e le classi 'Requests' e 'Data Boxes Of Productions'.

- Associazione tra 'Requests', 'Outputs Of Productions' e 'Data Types'.

  Con questa relazione ternaria si definisce che, per ogni richiesta, è necessario specificare il tipo dei file prodotti per ogni output della corrispondente produzione. L'attributo 'Medium' viene usato per indicare dove questi file devono essere memorizzati (su disk buffer, su nastro, o su una "directory privata"). L'attributo 'Target' assume valori diversi a seconda del valore dell'attributo 'Medium': se 'Medium' è pari a 'nastro', se ne definisce il corrispondente tipo; se invece 'Medium' specifica che i file di questo output devono essere memorizzati in una directory privata, questo attributo viene usato per definire il relativo pathname.

  Anche in questo caso per l'effettiva implementazione si può considerare un'ulteriore classe, con campi pari a 'Medium' e 'Target', in relazione con le altre tre classi.

In figura 6.29 sono rappresentate le associazioni tra le classi 'Stream Inputs Of Productions', 'Singleton Inputs Of Productions', 'Singleton Outputs Of Productions', 'Stream Outputs Of Productions', 'Data Files', 'Requests' e 'Elementary Requests'.

- Associazione tra 'Requests' e 'Elementary Requests'.

  La relazione rappresenta il fatto che le richieste vengono "scomposte" in elementary request.

  La relazione diretta è multivalore (una richiesta può essere scomposta in più elementary request) e parziale. La relazione inversa è univoca e totale (una elementary request fa riferimento ad un'unica richiesta).

- Associazione tra 'Singleton Inputs Of Productions', 'Requests' e 'Data Files'.

  Con questa associazione ternaria si definisce che per ogni richiesta, per ogni input di tipo singleton della corrispondente produzione, è associato un file (un singleton input file).

  Anche per questa relazione ternaria, e per quelle descritte in seguito, valgono i discorsi già fatti, ovvero l'introduzione di una quarta classe per l'effettiva implementazione della relazione ternaria.

- Associazione tra 'Stream Inputs Of Productions', 'Elementary Requests' e 'Data Files'.

  Questa associazione ternaria indica che per ogni elementary request, all'eventuale input di tipo stream della corrispondente production, è associato un file di dati (uno stream input file), che deve essere elaborato.

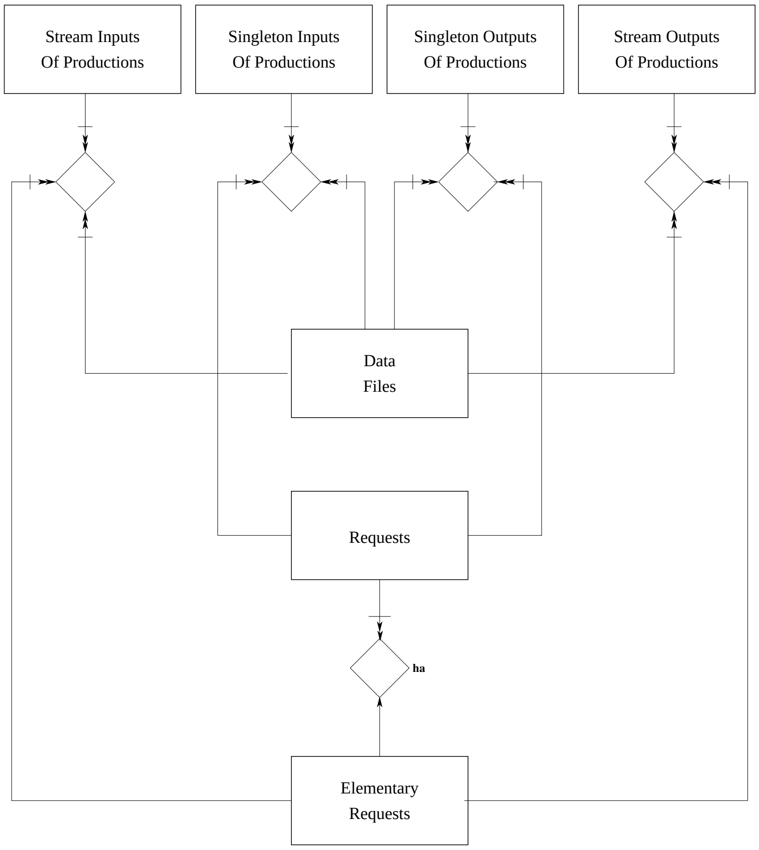


Figura 6.29: Schema della base di dati (parte V)

- Associazione tra 'Singleton Outputs Of Productions', 'Requests' e 'Data Files'.

  Questa relazione ternaria definisce che per ogni richiesta, per ogni output di tipo singleton della corrispondente produzione viene prodotto un file (un singleton output file).

- Associazione tra 'Stream Outputs Of Productions', 'Elementary Requests' e 'Data Files'.

  Questa associazione specifica che per ogni elementary request, per ogni output di tipo stream della corrispondente production viene generato un file di dati (uno stream output file).

In figura 6.30 sono rappresentate le associazioni tra le classi 'Local Elementary Processings', 'Processings Of Single Tasks', 'Tasks Of Productions', e 'Local Computers'.

- Associazione tra 'Local Elementary Processings' e 'Processings Of Single Tasks'.

  La relazione indica che ad ogni elementary request processata localmente sono associati i processing dei task che compongono la corrispondente production.

  La relazione diretta è multivalore (ad un elementary processing possono essere associati più processing di singoli task, nel caso in cui la corrispondente production sia costituita da più task o anche, in presenza di un singolo task, se è stato necessario elaborarlo più volte) e parziale (potrebbero non esserci processing di singoli task associati ad un elementary processing, ad esempio perché non si è ancora elaborato nessun task componente la relativa produzione). La relazione inversa è univoca e totale (un processing di un singolo task è associato ad un singolo elementary processing).

- Associazione tra 'Processings Of Single Tasks' e 'Tasks Of Productions'.

  Si indica che, per ogni elementary request, i task componenti la produzione vengono elaborati.

  La relazione diretta è univoca e totale (un processing è associato ad un singolo task). La relazione inversa è multivalore (ad un task componente di una production possono essere associati più processing) e parziale (potrebbero non esserci processing per un task componente di una production).

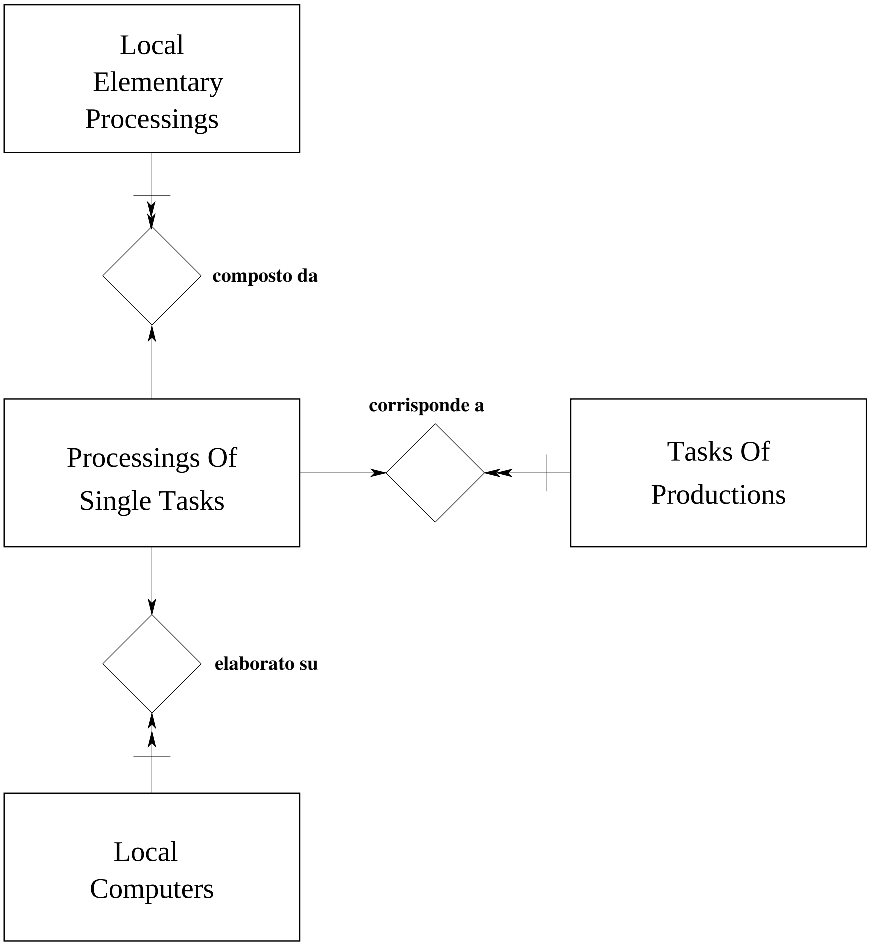


Figura 6.30: Schema della base di dati (parte VI)

- Associazione tra 'Processings Of Single Tasks' e 'Local Computers'.

  La relazione rappresenta il fatto che il processing di un task è eseguito su un computer.

  La relazione diretta è univoca e totale (il processing di un task è fatto su un singolo computer). La relazione inversa è multivalore (un computer può avere elaborato più task) e parziale (un computer potrebbe non avere processato nessun task).

6.4 Rappresentazione con il modello relazionale

Come descritto nel capitolo 8, nell'implementazione del primo prototipo si è utilizzato un database relazione. Lo schema della base di dati rappresentato con i costrutti del modello a oggetti è stato quindi trasformato in uno schema relazionale.

Sostanzialmente per rappresentare una classe nel modello relazionale si considera una relazione in cui si riportano gli attributi della classe considerata. Si considerano poi le seguenti regole [6]:

- un'associazione tra una classe A e una classe B, con diretta multivalore e inversa univoca, viene definita nel modello relazionale attraverso gli schemi R_A e R_B, con R_A associato alla classe A, e R_B associato alla classe B. Agli attributi di R_B si aggiunge una chiave esterna che riferisce la relazione R_A;

- per un'associazione tra una classe A e una classe B, con diretta e inversa univoca, la chiave esterna si aggiunge ad una qualunque delle relazioni, preferendo quella rispetto a cui l'associazione è totale;

- un'associazione tra una classe A e una classe B, con diretta e inversa multivalore, viene definita nel modello relazionale attraverso le relazioni R_A (associata alla classe A), R_B (associata alla classe B), e da una terza relazione, che contiene due chiavi esterne che riferiscono le due relazioni coinvolte;

- per rappresentare un'associazione tra n classi, si considera un'ulteriore relazione che contiene le chiavi esterne.

Per questi due ultimi casi considerati, se l'associazione ha degli attributi, questi vengono aggiunti alla nuova relazione.

Vengono di seguito descritti gli schemi di relazione della base di dati per il sistema DEMI.

Attributo	Tipo	Descrizione
ArchitectId	String	L'identificatore dell'architettura
Description	String	La descrizione dell'architettura
OtherInfo	String	Altre eventuali informazioni

Tabella 6.1: Schema di relazione 'Architectures'

ArchitectId	Description	OtherInfo
win32	Computer con processore Intel OS: Windows 32 bit	Windows NT e Windows 95
axp-win32	Computer con processore Alpha OS: Windows 32 bit	Windows NT e Windows 95
axp-du	Computer con processore Alpha OS: Digital Unix	
hp-ux	Workstation HP OS: HP-UX 10.20	
...	...	...

Tabella 6.2: Esempio di relazione

6.4.1 Architetture

La tabella 6.1 definisce lo schema di relazione *Architectures*, usato per rappresentare la classe 'Architectures' del modello a oggetti.

La chiave primaria di questa tabella è costituita dall'attributo *ArchitectId*.

In tabella 6.2 è mostrato un esempio di istanza di questo schema.

6.4.2 Modelli di tape device

Per definire la classe 'Tape Device Models' del modello a oggetti si considera lo schema di relazione *TapeDeviceModels*, rappresentato in tabella 6.3.

La chiave primaria è rappresentata dall'attributo *TapeDevModId*.

Un esempio di relazione è riportato in tabella 6.4.

6.4.3 Global administrator

La tabella 6.5 definisce lo schema di relazione *GlobalAdministrators*, utilizzato per rappresentare la classe 'Global Administrators' del modello a oggetti.

Attributo	Tipo	Descrizione
TapeDevModId	String	L'identificatore del modello di tape device
Description	String	La descrizione del modello di tape device
Capacity	Real	La capacità (espressa ad esempio in MB) di un nastro di questo modello
OtherInfo	String	Altre eventuali informazioni riguardanti il modello di tape device

Tabella 6.3: Schema di relazione 'TapeDeviceModels'

TapeDevModId	Description	Capacity	OtherInfo
dlt	DLT, per nastri TZ87	30000	
exab	Exabyte	3500	
...	...	...	...

Tabella 6.4: Esempio di relazione

Attributo	Tipo	Descrizione
UserId	String	L'identificatore del global administrator
Password	String	La password (criptata) del global administrator
EMailAddr	String	L'e-mail address del global administrator
OtherInfo	String	Altre eventuali informazioni riguardanti il global administrator

Tabella 6.5: Schema di relazione 'GlobalAdministrators'

UserId	Password	EMailAddr	OtherInfo
globadm	d51e@qeg78s	globadm@domain	Default global administrator
sgaravat	d3+egk*y3s	sgaravat@pd.infn.it	Massimo Sgaravatto Tel. ++39 49 8277029
rossi	3efgy3s	rossi@domain	Mario Rossi Tel. ++39 49 123456
...	...	...	...

Tabella 6.6: Esempio di relazione

Attributo	Tipo	Descrizione
CompName	String	Il nome o l'indirizzo IP del computer
Description	String	La descrizione del computer
ArchitectId	String	L'identificatore che specifica l'architettura del computer
Perform	Real	Il parametro che definisce le prestazioni del computer
Status	Boolean	Indica se il computer è 'abilitato' o 'disabilitato'

Tabella 6.7: Schema di relazione 'LocalComputers'

La chiave primaria di questo schema di relazione è rappresentata dall'attributo *UserId*.

Un esempio di relazione è riportato in tabella 6.6.

6.4.4 Computer

Per definire la classe 'Local Computers' del modello a oggetti si considera uno specifico schema di relazione, rappresentato in tabella 6.7, chiamato *LocalComputers*.

La chiave primaria è costituita dall'attributo *CompName*.

Con l'attributo *ArchitectId* si rappresenta l'associazione tra le classi 'Local Computers' e 'Architectures'.

Un'istanza di relazione relativa a questo schema è mostrata in tabella 6.8.

CompName	Description	ArchitectId	Perform	Status
osde00	Alpha Server 1000	axp-du	100	Enabled
hpcms01		hp-ux	105	Enabled
pccms1	Pentium Pro 200	win32	75	Enabled
...	...	...	...	...

Tabella 6.8: Esempio di relazione

Attributo	Tipo	Descrizione
BufferId	String	L'identificatore del disk buffer
CompName	String	Il nome del computer dove il disco è installato
TotSp	Real	Lo spazio totale (espresso ad esempio in MB) assegnato per la memorizzazione di file DEMI
UsSp	Real	Lo spazio (espresso ad esempio in MB) già usato in questo buffer
FreeSp	Real	Lo spazio (espresso ad esempio in MB) ancora disponibile in questo buffer
Status	Boolean	Indica se il buffer è 'abilitato' o 'disabilitato'
OtherInfo	String	Altre eventuali informazioni

Tabella 6.9: Schema di relazione 'LocalBuffers'

6.4.5 Buffer

Lo schema di relazione *LocalBuffers*, rappresentato in tabella 6.9, definisce la classe 'Local Buffers' del modello a oggetti.

La chiave primaria di questo schema di relazione è costituita dall'attributo *BufferId*.

L'attributo *CompName* è la chiave esterna che permette di definire l'associazione tra le classi 'Local Computers' e 'Local Buffers'.

La tabella 6.10 mostra un esempio di relazione.

In tabella 6.11 è rappresentato lo schema di relazione *BufferPathnames*, usato per definire l'associazione con attributo tra le classi 'Local Buffers' e 'Architectures'; un esempio di istanza di questo schema è mostrato in tabella 6.12.

BufferId	CompName	TotSp	UsSp	FreeSp	Status	OtherInfo
buffer1	osde00	1000	570	300	Enabl	
buffer2	osde00	2000	1300	700	Enabl	
buffer3	hpcms01	500	300	113	Enabl	
...	...	...	...	...	...	...

Tabella 6.10: Esempio di relazione

Attributo	Tipo	Descrizione
BufferId	String	L'identificatore del disk buffer
ArchitectId	String	L'identificatore dell'architettura
PathName	String	Il pathname del buffer

Tabella 6.11: Schema di relazione 'BufferPathnames'

BufferId	ArchitectId	Pathname
buffer1	axp-du	/DEMI/Buffer1
buffer1	win32	%DEMIBUFFER1%
buffer2	axp-du	/DEMI/Buffer1
buffer2	win32	%DEMIBUFFER2%
...	...	...

Tabella 6.12: Esempio di relazione

Attributo	Tipo	Descrizione
TapeDevId	String	L'identificatore del tape device
TapeDevModId	String	L'identificatore del modello di tape device
CompName	String	Il nome del computer dove il tape device è installato
PathName	String	Il pathname con cui si "accede" al tape device
Status	Boolean	Indica se il tape device è 'abilitato' o 'disabilitato'
OtherInfo	String	Altre eventuali informazioni

Tabella 6.13: Schema di relazione 'LocalTapeDevices'

TapeDevId	TapeDevModId	CompName	PathName	Status	OtherInfo
tapedev1	dlt	osde00	/dev/rmt0h	Enabled	
tapedev2	dlt	hpcms01	/dev/rmt0h	Enabled	
tapedev3	exab	hpcms01	/dev/rmt1h	Enabled	
...	...	...	...	...	...

Tabella 6.14: Esempio di relazione

6.4.6 Tape Device

In tabella 6.13 è definito lo schema di relazione *LocalTapeDevices*, che permette di rappresentare la corrispondente classe del modello a oggetti.

La chiave primaria è definita dall'attributo *TapeDevId*.

L'attributo *TapeDevModId* permette di definire l'associazione tra le classi 'Local Tape Devices' e 'Tape Device Models', mentre con l'attributo *CompName* si specifica la relazione tra le classi 'Local Tape Devices' e 'Local Computers'.

La tabella 6.14 rappresenta un esempio di relazione.

6.4.7 Utenti

Per definire la classe 'Users' e le sue sottoclassi 'Local Administrators', 'Operators' e 'General Users', si considera lo schema di relazione *Users*, rappresentato in tabella 6.15.

La chiave primaria di questo schema è costituita dall'attributo *UserId*.

Un esempio di relazione, relativo a questo schema, è riportata in figura 6.16.

Attributo	Tipo	Descrizione
UserId	String	Lo user identifier dell'utente
Password	String	La password (criptata) dell'utente
EMailAddr	String	L' e-mail address dell'utente
LocAdm	Boolean	Definisce se l'utente è un local administrator
Operat	Boolean	Specifica se l'utente è un operator
GenUser	Boolean	Definisce se l'utente è un general user
OtherInfo	String	Altre eventuali informazioni riguardanti l'utente

Tabella 6.15: Schema di relazione 'Users'

UserId	Password	EMailAddr	LocAdm	Operat	GenUser	OtherInfo
locadm	wdd@t43	locad@domain	Yes	No	No	Default Local Administrator
operator	w!45f67u	oper@domain	No	Yes	No	Default Operator
rossi	3efg*y3s	rossi@domain	No	Yes	Yes	Mario Rossi Tel. 123456
neri	326g)yds	neri@domain	No	No	Yes	Luca Neri Tel. 314436
...	...	...	...	...	...	...

Tabella 6.16: Esempio di relazione

Attributo	Tipo	Descrizione
FileId	String	L'identificatore del file
Description	String	La descrizione del tipo di dati rappresentati dal file
OtherInfo	String	Altre eventuali informazioni

Tabella 6.17: Schema di relazione 'DataFiles'

FileId	Description	OtherInfo
id00001	QQPS 183 GeV DSIM97-1	
id00002	BAFO 172 GeV DSIM96-1	
id00003	QQPS 183 GeV DSIM97-1	
...	...	...

Tabella 6.18: Esempio di relazione

6.4.8 Dati

Lo schema di relazione *DataFiles*, rappresentato in tabella 6.17, definisce la corrispondente classe del modello a oggetti.

La chiave primaria è costituita dall'attributo *FileId*.

L'attributo *Description* è la chiave esterna che permette di definire l'associazione tra le classi 'Data Files' e 'Data Types'.

Un esempio di istanza dello schema è mostrato in tabella 6.18.

Lo schema di relazione *LocalDataFiles*, mostrato in tabella 6.19, permette di rappresentare la classe 'Local Data Files'.

La chiave primaria di questo schema di relazione è definita dall'attributo *FileId*.

Un esempio di relazione, considerando questo schema, è rappresentato in tabella 6.20.

Attributo	Tipo	Descrizione
FileId	String	L'identificatore del file
Size	Real	La dimensione (espressa ad esempio in MB) del file
OtherInfo	String	Altre eventuali informazioni

Tabella 6.19: Schema di relazione 'LocalDataFiles'

FileId	Size	OtherInfo
id00001	100	
id00002	135	
...	...	...

Tabella 6.20: Esempio di relazione

Attributo	Tipo	Descrizione
TapeId	String	L'identificatore del nastro
FreeSpac	Real	Lo spazio ancora disponibile sul nastro
TapeDevModId	String	Il modello del nastro
Description	String	Il tipo dei file memorizzati nel nastro
OtherInfo	String	Altre eventuali informazioni

Tabella 6.21: Schema di relazione 'LocalTapes'

In tabella 6.21 è mostrato lo schema di relazione *LocalTapes*, con cui si definisce la relativa classe del modello a oggetti.

La chiave primaria è costituita dall'attributo *TapeId*.

L'attributo *TapeDevModId* permette di definire l'associazione tra le classi 'Local Tapes' e 'Tape Device Models'. Per rappresentare la relazione tra le classi 'Local Tapes' e 'Data Types', si usa invece l'attributo *Description*.

La tabella 6.22 mostra un esempio di relazione.

Lo schema di relazione *FilesOnBuffers*, definito in tabella 6.23, è utilizzato per definire l'associazione tra le classi 'Local Buffers' e 'Local Data Files' del modello a oggetti.

Un esempio di relazione, con riferimento a questo schema, è rappresentato in tabella 6.24.

TapeId	FreeSpace	TapeDevModId	Description	OtherInfo
tapeid1	900	dlt	QQPS 183 GeV DSIM97-1	
tapeid2	1200	exab	BAFO 172 GeV DSIM96-1	
tapeid3	2680	dlt	QQPS 183 GeV DSIM97-1	
...	...	...	...	...

Tabella 6.22: Esempio di relazione

Attributo	Tipo	Descrizione
FileId	String	L'identificatore del file
BufferId	String	L'identificatore del disk buffer

Tabella 6.23: Schema di relazione 'FilesOnBuffers'

FileId	BufferId
id00001	buffer1
id00001	buffer2
id00002	buffer1
...	...

Tabella 6.24: Esempio di relazione

Per definire la relazione tra la classe 'Local Data Files' e 'Local Tapes', si specifica uno schema di relazione chiamato *FilesOnTapes*, rappresentato in tabella 6.25.

In tabella 6.26 è presentato un esempio di istanza di relazione.

6.4.9 Task

Lo schema *TaskTypes*, rappresentato in tabella 6.27, permette di definire la corrispondente classe del modello a oggetti.

La chiave primaria di questo schema è definita dall'attributo *TaskId*.

L'attributo *UserId* è la chiave esterna che definisce la relazione tra le classi 'Task Types' e 'General Users'.

Un esempio di istanza, corrispondente a questo schema di relazione, è mostrato in tabella 6.28.

In tabella 6.29 è definito lo schema di relazione *IOsOfTasks*, usato per rappresentare la classe 'IOs Of Tasks', e le relative sottoclassi.

La chiave primaria di questo schema è costituita dall'attributo *IOId*.

Attributo	Tipo	Descrizione
FileId	String	L'identificatore del file
TapeId	String	L'identificatore del nastro

Tabella 6.25: Schema di relazione 'FilesOnTapes'

FileId	TapeId
id00001	tapeid1
id00001	tapeid2
id00002	tapeid1
...	...

Tabella 6.26: Esempio di relazione

Attributo	Tipo	Descrizione
TaskId	String	L'identificatore del tipo di task
Description	String	La descrizione del tipo di task
UserId	String	L'identificatore dell'utente che ha definito questo tipo di task
OtherInfo	String	Altre eventuali informazioni

Tabella 6.27: Schema di relazione 'TaskTypes'

TaskId	Description	UserId	OtherInfo
tasktype1	Task che esegue l'elaborazione X	rossi	
tasktype2	Task che esegue l'elaborazione Y	bianchi	
tasktype3	Task che esegue l'elaborazione Z	rossi	
...	...	...	...

Tabella 6.28: Esempio di relazione

Attributo	Tipo	Descrizione
IOId	String	L'identificatore dell'input/output
Name	String	Il nome dell'input/output
TaskId	String	L'identificatore del tipo di task a cui questo input/output fa riferimento
IO	Boolean	L'indicazione se si tratta di un input o di un output
ParNo	Integer	Il numero di parametro con cui si identifica l'input/output nella riga di comando che "invoca" l'esecuzione del corrispondente task

Tabella 6.29: Schema di relazione 'IOsOfTasks'

IOId	Name	TaskId	IO	ParNo
tasktype1.i1	i1	tasktype1	input	1
tasktype1.i2	i2	tasktype1	input	2
tasktype1.i3	i3	tasktype1	input	3
tasktype1.o1	o1	tasktype1	output	4
tasktype1.o2	o2	tasktype1	output	5
tasktype2.i1	i1	tasktype2	input	1
tasktype2.i2	i2	tasktype2	input	2
tasktype2.o1	o1	tasktype2	output	3
tasktype3.i1	i1	tasktype3	input	1
tasktype3.i2	i2	tasktype3	input	2
tasktype3.o1	o1	tasktype3	output	3
tasktype3.o2	o2	tasktype3	output	4
...	...	...	...	...

Tabella 6.30: Esempio di relazione

L'attributo *TaskId* permette di definire l'associazione tra la classe 'Task Types' e le classi che definiscono i corrispondenti input-output.

Con riferimento ai tipi di task rappresentati in figura 6.31, in tabella 6.30 è presentato un esempio di relazione.

Lo schema di relazione *ImplementationsOfTasks* è usato per definire la relativa classe del modello a oggetti. Questo schema è rappresentato in tabella 6.31.

La chiave primaria è rappresentata dall'attributo *TaskImplemId*.

L'attributo *TaskId* permette di definire l'associazione tra le classi 'Implementations Of Tasks' e 'Task Types', mentre l'attributo *ArchitectId* viene usato per specificare la relazione tra le classi 'Implementations Of Tasks' e 'Architectures'. Infine con l'attributo *ExecFile* si definisce l'associazione tra le classi 'Implementations Of Tasks' e 'Executable Files'.

In tabella 6.32 è mostrato un esempio di istanza di questo schema.

Lo schema di relazione *FilesForImplementations*, rappresentato in tabella 6.33, consente di definire la classe 'Files For Implementations' del modello a oggetti, e l'associazione tra questa classe e la classe 'Implementations Of Tasks'.

Un esempio di relazione è riportato in tabella 6.34.

Lo schema *ParametersForTasks*, definito in tabella 6.35, viene utilizzato per rappresentare la corrispondente classe del modello a oggetti.

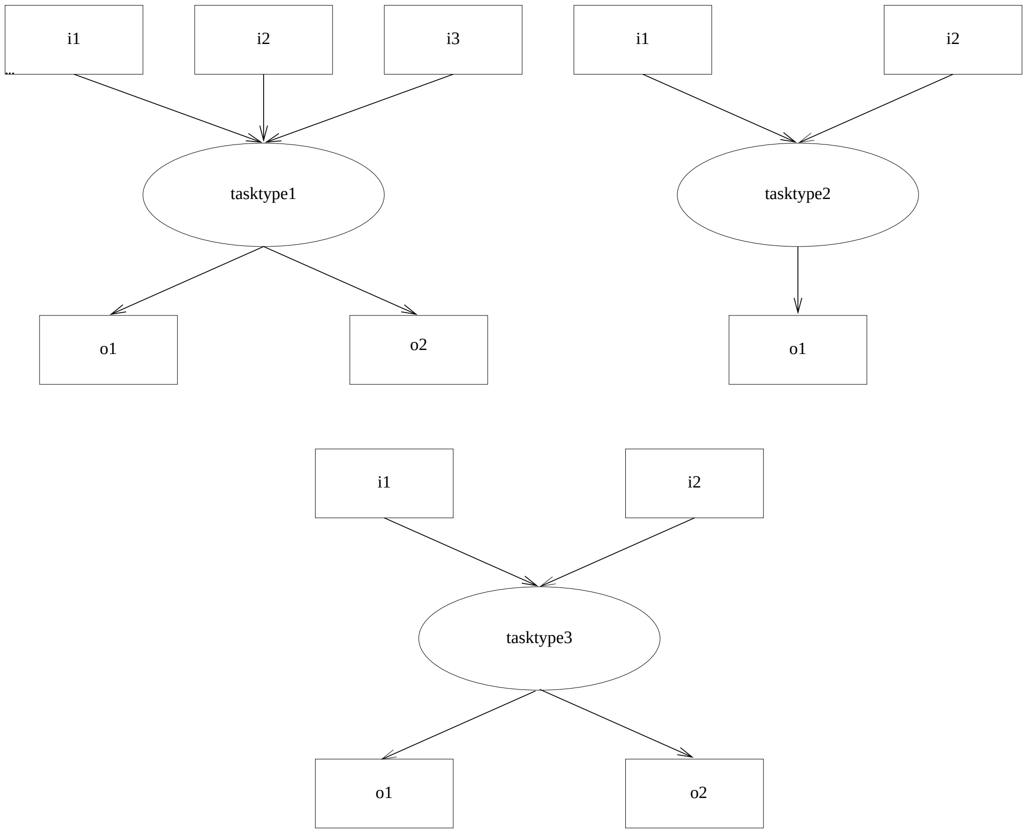


Figura 6.31: Esempi di tipi di task

Attributo	Tipo	Descrizione
TaskImplemId	String	L'identificatore dell'implementazione
TaskId	String	L'identificatore del tipo di task
ArchitectId	String	L'identificatore dell'architettura
ExecFile	String	Il nome del file che deve essere specificato nella riga di comando per l'esecuzione
Cost	Real	Il parametetro 'Cost of the task'
OtherInfo	String	Altre eventuali informazioni

Tabella 6.31: Schema di relazione 'ImplementationsOfTasks'

TaskImplemId	TaskId	ArchitectId	ExecFile	Cost
tasktype1.win32	tasktype1	win32	$DEMI/TaskType1/ win32/start.bat	5
tasktype1.axp-du	tasktype1	axp-du	$DEMI/TaskType1/ axp-du/start.sh	5
tasktype2.hp-ux	tasktype2	hp-ux	$DEMI/TaskType2/ hp-ux/start.sh	5
tasktype2.axp-win32	tasktype2	axp-win32	$DEMI/TaskType2/ axp-win32/start.bat	5
tasktype3.hp-ux	tasktype3	hp-ux	$DEMI/TaskType3/ hp-ux/start.sh	5
...	...	...	...	...

Tabella 6.32: Esempio di relazione

Attributo	Tipo	Descrizione
TaskImplemId	String	L'identificatore dell'implementazione del tipo di task
FileName	String	Il pathname del file

Tabella 6.33: Schema di relazione 'FilesForImplementations'

TaskImplemId	FileName
tasktype1.win32	$DEMI/TaskType1/win32/start.bat
tasktype1.win32	$DEMI/TaskType1/win32/execfile.exe
tasktype1.win32	$DEMI/TaskType1/win32/titlefile.tit
tasktype1.axp-du	$DEMI/TaskType1/axp-du/start.sh
tasktype1.axp-du	$DEMI/TaskType1/axp-du/execfile
tasktype1.axp-du	$DEMI/TaskType1/axp-du/titlefile.tit
tasktype2.hp-ux	$DEMI/TaskType2/hp-ux/start.sh
tasktype2.hp-ux	$DEMI/TaskType2/hp-ux/execfile
tasktype2.axp-win32	$DEMI/TaskType2/axp-win32/start.bat
tasktype2.axp-win32	$DEMI/TaskType2/axp-win32/execfile.exe
tasktype3.hp-ux	$DEMI/TaskType3/hp-ux/start.sh
tasktype3.hp-ux	$DEMI/TaskType3/hp-ux/execfile
tasktype3.hp-ux	$DEMI/TaskType3/hp-ux/titlefile.tit
...	...

Tabella 6.34: Esempio di relazione

Attributo	Tipo	Descrizione
TaskImplemId	String	L'identificatore dell'implementazione del tipo di task
ParNo	Integer	Il numero ordinale del parametro (primo/secondo/...)
Value	String	Il valore associato al parametro

Tabella 6.35: Schema di relazione 'ParametersForTasks'

TaskImplemId	ParNo	Value
tasktype1.win32	6	100
tasktype2.hp-ux	4	$DEMI/TaskType2/hp-ux/title.tit
tasktype2.hp-ux	5	200
...	...	...

Tabella 6.36: Esempio di relazione

Attributo	Tipo	Descrizione
ProdId	String	L'identificatore della produzione
Description	String	La descrizione della produzione
UserId	String	L'identifier dell'utente che ha definito la produzione
Status	Boolean	Indica se la produzione può essere usata per sottomettere richieste
OtherInfo	String	Altre eventuali informazioni

Tabella 6.37: Schema di relazione 'Productions'

Utilizzando la chiave esterna *TaskImplemId* si rappresenta l'associazione tra le classi 'Implementations Of Tasks' e 'Parameters For Tasks'.

In tabella 6.36 è mostrato l'esempio di un'istanza dello schema.

6.4.10 Production

In tabella 6.37 è mostrato lo schema di relazione *Productions*, utilizzato per la rappresentazione della relativa classe del modello a oggetti.

La chiave primaria è costituita dall'attributo *ProdId*.

L'attributo *UserId* permette di "modellare" l'associazione tra le classi 'General Users' e 'Productions'.

In tabella 6.38 è mostrato un esempio di istanza per lo schema considerato.

La classe 'Tasks Of Productions' del modello a oggetti viene rappresentata attraverso lo schema di relazione *TasksOfProductions*, definito in tabella 6.39.

La chiave primaria è rappresentata dall'attributo *TaskOfProdId*.

L'attributo *TaskId* è la chiave esterna che definisce la relazione tra le classi 'Tasks Of Productions' e 'Task Types', mentre l'associazione tra le classi 'Tasks Of Productions' e 'Productions' è rappresentata grazie alla chiave esterna *ProdId*.

ProdId	Description	UserId	Status	OtherInfo
prod1	Produzione che gestisce l'attività X	rossi	available	
prod2	Produzione che gestisce l'attività Y	bianchi	available	
prod3	Produzione che gestisce l'attività Z	rossi	not available	
prod4	Produzione che gestisce l'attività W	rossi	not available	
...	...	...	...	...

Tabella 6.38: Esempio di relazione

Attributo	Tipo	Descrizione
TaskOfProdId	String	L'identificatore del task componente di una produzione
Name	String	Nome del task
TaskId	String	L'identificatore del tipo di task
ProdId	String	L'identificatore della produzione

Tabella 6.39: Schema di relazione 'TasksOfProductions'

TaskOfProdId	Name	TaskId	ProdId
prod1.t1	t1	tasktype1	prod1
prod1.t2	t2	tasktype2	prod1
prod1.t3	t3	tasktype3	prod1
prod2.t1	t1	tasktype2	prod2
...	...	...	...

Tabella 6.40: Esempio di relazione

Attributo	Tipo	Descrizione
DataBoxId	String	L'identificatore del data box di una produzione
Name	String	Nome del data box
ProdId	String	L'identificatore della corrispondente production
Type	Boolean	L'indicazione se si tratta di un data box di tipo stream o di tipo singleton
Role	String	"Ruolo" di questo data box (input/output/temporary)

Tabella 6.41: Schema di relazione 'DataBoxesOfProductions'

Con riferimento alle produzioni rappresentate in figura 6.32, definite attraverso i tipi di task prima mostrati in figura 6.31, in tabella 6.40 è riportato un esempio di relazione.

Lo schema *DataBoxesOfProductions*, definito in tabella 6.41, rappresenta la classe 'Data Boxes of Productions' del modello a oggetti, e le relative sottoclassi.

La chiave primaria di questo schema è definito dall'attributo *DataBoxId*.

Attraverso la chiave esterna *ProdId* si definisce la relazione tra le classi 'Data Boxes Of Productions' e 'Productions'.

Considerando sempre le produzioni di figura 6.32, in tabella 6.42 è mostrato un esempio di relazione.

Lo schema di relazione *Relations*, definito in tabella 6.43, permette di rappresentare l'associazione ternaria tra le classi 'Tasks Of Productions', 'Data Boxes Of Productions' e 'IOs Of Tasks'.

L'istanza di relazione relativa alle produzioni di figura 6.32 è presentata in tabella 6.44

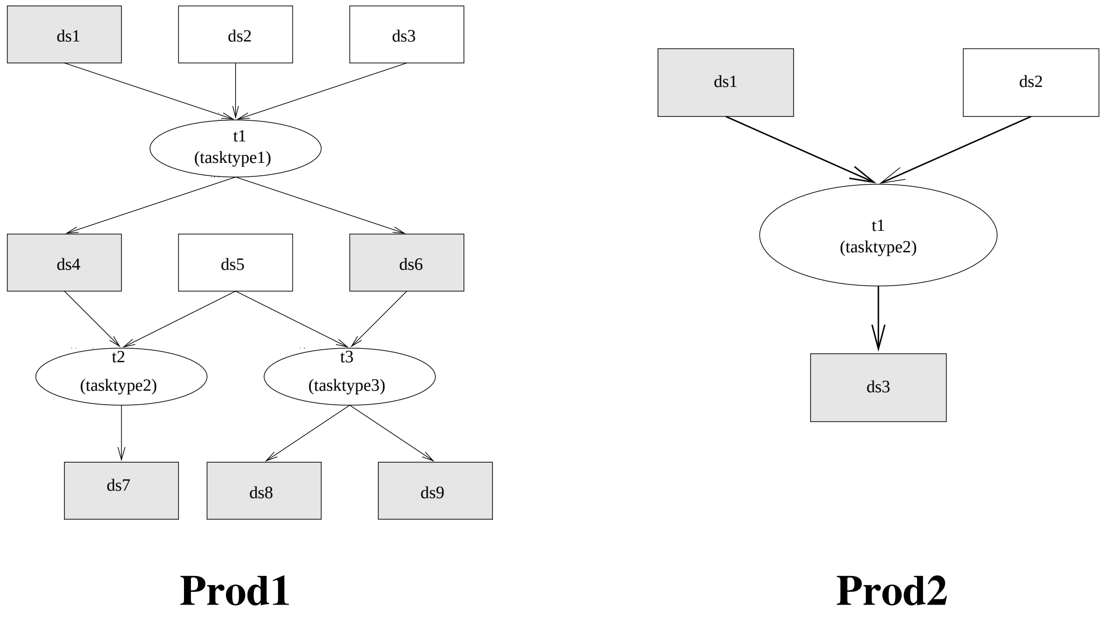


Figura 6.32: Esempi di produzioni

DataBoxId	Name	ProdId	Type	Role
prod1.ds1	ds1	prod1	stream	input
prod1.ds2	ds2	prod1	singleton	input
prod1.ds3	ds3	prod1	singleton	input
prod1.ds4	ds4	prod1	stream	output
prod1.ds5	ds5	prod1	singleton	input
prod1.ds6	ds6	prod1	stream	temp
prod1.ds7	ds7	prod1	stream	output
prod1.ds8	ds8	prod1	stream	output
prod1.ds9	ds9	prod1	stream	output
prod2.ds1	ds1	prod2	stream	input
prod2.ds2	ds2	prod2	singleton	input
prod2.ds3	ds3	prod2	stream	output
...	...	...	..	... .

Tabella 6.42: Esempio di relazione

Attributo	Tipo	Descrizione
DataBoxId	String	L'identificatore di un data box di una produzione
IOId	String	L'identificatore di un input/output di un tipo di task
TaskOfProdId	String	L'identificatore di un task componente di una produzione

Tabella 6.43: Schema di relazione 'Relations'

DataBoxId	IOId	TaskOfProdId
prod1.ds1	tasktype1.i1	prod1.t1
prod1.ds2	tasktype1.i2	prod1.t1
prod1.ds3	tasktype1.i3	prod1.t1
prod1.ds4	tasktype1.o1	prod1.t1
prod1.ds4	tasktype2.i1	prod1.t2
prod1.ds5	tasktype2.i2	prod1.t2
prod1.ds5	tasktype3.i1	prod1.t3
prod1.ds6	tasktype1.o2	prod1.t1
prod1.ds6	tasktype3.i2	prod1.t3
prod1.ds7	tasktype2.o1	prod1.t2
prod1.ds8	tasktype3.o1	prod1.t3
prod1.ds9	tasktype3.o2	prod1.t3
prod2.ds1	tasktype2.i1	prod2.t1
prod2.ds2	tasktype2.i2	prod2.t1
prod2.ds3	tasktype2.o1	prod2.t1

Tabella 6.44: Esempio di relazione

Attributo	Tipo	Descrizione
ReqId	String	L'identificatore della richiesta
UserId	String	L'identificatore dell'utente che ha sottomesso la richiesta
ProdId	String	L'identificatore della produzione
Prior	Integer	Il livello di priorità della richiesta
SubmitTime	Date-Time	La data e l'ora in cui la richiesta è stata sottomessa
NumIt	Integer	Il numero di volte per cui la relativa produzione deve essere eseguita (se non ci sono stream input)
Status	String	Lo stato dell'elaborazione
OtherInfo	String	Altre eventuali informazioni

Tabella 6.45: Schema di relazione 'Requests'

ReqId	UserId	ProdId	Prior	SubmitTime	NumIt	Status	OtherInfo
req001	rossi	prod1	5	21-03-1998 13:42	0	Satisfied	
req002	bianchi	prod2	4	21-03-1998 15:13	0	In execution	
...	...	...	...	...	...	...	...

Tabella 6.46: Esempio di relazione

6.4.11 Computational request

Lo schema di relazione *Requests*, rappresentato in tabella 6.45, viene utilizzato per definire la relativa classe del modello a oggetti.

La chiave primaria di questo schema di relazione è definita dall'attributo *ReqId*.

L'attributo *UserId* definisce l'associazione tra le classi 'General Users' e 'Requests', mentre con l'attributo *ProdId* si rappresenta l'associazione tra le classi 'Requests' e 'Productions'.

Un esempio di istanza di questo schema di relazione è rappresentato in tabella 6.46.

L'associazione con attributi tra le classi 'Requests' e 'Data Boxes Of Productions' viene rappresentata attraverso lo schema di relazione *DataBoxesOfRequests*, rappresentato in tabella 6.47.

Una possibile istanza relativa a questo schema, è riportata in tabella 6.48

Attributo	Tipo	Descrizione
ReqId	String	L'identificatore della richiesta
DataBoxId	String	L'identificatore del data box della produzione
MaxSize	Real	La dimensione massima (espressa ad esempio in MB) di un file che verrà associato al data box, per la richiesta considerata

Tabella 6.47: Schema di relazione 'DataBoxesOfRequests'

ReqId	DataBoxId	MaxSize
req001	prod1.ds1	20
req001	prod1.ds2	2
req001	prod1.ds3	2
req001	prod1.ds4	120
req001	prod1.ds5	3.5
req001	prod1.ds6	85
req001	prod1.ds7	25
req001	prod1.ds8	50
req002	prod2.ds1	100
req002	prod2.ds2	1
req002	prod2.ds3	10
...	...	...

Tabella 6.48: Esempio di relazione

Attributo	Tipo	Descrizione
ReqId	String	L'identificatore della richiesta
DataBoxId	String	L'identificatore del data box della produzione
Description	String	Il tipo dei file prodotti con questo output
Medium	String	Il "supporto" da utilizzare per la memorizzazione dei file corrispondenti a questo output (buffer, nastro, directory "privata")
Target	String	Il nome della directory privata, o il tipo di nastro (a seconda del valore dell'attributo Medium)

Tabella 6.49: Schema di relazione 'DescriptionsOfOutputsOfRequests'

ReqId	DataBoxId	Description	Medium	Target
req001	prod1.ds4	QQPS ZEB971	buffer	
req001	prod1.ds7	QQPS DST971	tape	dlt
req001	prod1.ds8	QQPS ANA971	tape	
req001	prod1.ds9	QQPS PTR971	tape	
req002	prod2.ds3	BAFO PHI971	privdir	$DISK1/Output1

Tabella 6.50: Esempio di relazione

Con lo schema di relazione *DescriptionsOfOutputsOfRequests*, mostrato in tabella 6.49, si "modella" l'associazione ternaria con attributi tra le classi 'Requests', 'Outputs Of Productions' e 'Data Types'.

In tabella 6.50 è presentato un esempio di istanza di relazione.

La relazione *SingletionIOOfRequests*, rappresentata in tabella 6.51, definisce le associazioni ternarie tra le classi 'Requests', 'Singleton Inputs Of Productions' e 'Data Files', e tra le classi 'Requests', 'Singleton Outputs Of Productions' e 'Data Files'.

Attributo	Tipo	Descrizione
ReqId	String	L'identificatore della richiesta
DataBoxId	String	L'identificatore del data box della produzione
FileId	String	L'identificatore del file

Tabella 6.51: Schema di relazione 'SingletonIOOfRequests'

ReqId	DataBoxId	FileId
req001	prod1.ds2	id00001
req001	prod1.ds3	id00005
req001	prod1.ds5	id00105
req002	prod2.ds2	id10113

Tabella 6.52: Esempio di relazione

Attributo	Tipo	Descrizione
ElemProcId	String	L'identificatore dell'elementary request
DataBoxId	String	L'identificatore del data box della produzione
FileId	String	L'identificatore del file

Tabella 6.53: Schema di relazione 'StreamIOOfElementaryRequests'

Un esempio di relazione, relativa a questo schema, è presentato in tabella 6.52.

Lo schema di relazione *StreamIOOfElementaryRequests*, rappresentato in tabella 6.53, definisce le associazioni ternarie tra le classi 'Elementary Requests', 'Stream Inputs Of Productions' e 'Data Files', e tra le classi 'Elementary Requests', 'Stream Outputs Of Productions' e 'Data Files'.

In tabella 6.54 è mostrato un esempio di relazione.

ElemProcId	DataBoxId	FileId
elproc1001	prod1.ds1	id20001
elproc1002	prod1.ds1	id20002
elproc1003	prod1.ds1	id20003
elproc2001	prod2.ds1	id31001
elproc2002	prod2.ds1	id31002
elproc2003	prod2.ds1	id31003
elproc2004	prod2.ds1	id31004
elproc2005	prod2.ds1	id31005

Tabella 6.54: Esempio di relazione

Attributo	Tipo	Descrizione
ElemReqId	String	L'identificatore della elementary request (e del relativo processing)
ReqId	String	L'identificatore della corrispondente richiesta
Status	String	Lo stato di esecuzione del processing
OtherInfo	String	Altre eventuali informazioni

Tabella 6.55: Schema di relazione 'ElementaryRequests'

ElemReqId	ReqId	Status	OtherInfo
elproc00001	req001	succesfully completed	
elproc00002	req001	in execution	
elproc00003	req002	aborted	
...	...	...	...

Tabella 6.56: Esempio di relazione

6.4.12 Processing

In tabella 6.55 è definito lo schema *ElementaryRequests*, che rappresenta la classe 'Elementary Requests' del modello a oggetti.

La chiave primaria è definita dall'attributo *ElemReqId*.

L'attributo *ReqId* è la chiave esterna che definisce l'associazione tra le classi 'Requests' e 'Elementary Requests'.

Un esempio di istanza per questo schema di relazione è rappresentato in tabella 6.56.

Per rappresentare la classe 'Local Elementary Processings' del modello a oggetti si utilizza lo schema di relazione *LocalElementaryProcessings*, definito in tabella 6.57.

La chiave primaria è costituita dall'attributo *ElemReqId*.

Attributo	Tipo	Descrizione
ElemReqId	String	L'identificatore dell'elementary request (e del relativo processing)
OtherInfo	String	Altre eventuali informazioni

Tabella 6.57: Schema di relazione 'LocalElementaryProcessings'

ElemReqId	OtherInfo
elp00001	
elp00003	
elp00013	
...	...

Tabella 6.58: Esempio di relazione

Attributo	Tipo	Descrizione
ElemReqId	String	L'identificatore dell'elementary request (e del relativo processing)
TaskOfProdId	String	L'identificatore del task componente la produzione
CompName	String	Il nome del computer a cui è stata assegnata l'esecuzione del task
Status	String	Lo stato dell'esecuzione
StartTime	Date-Time	La data e l'ora in cui è iniziata l'esecuzione
EndTime	Date-Time	La data e l'ora in cui è terminata l'esecuzione

Tabella 6.59: Schema di relazione 'ProcessingsOfSingleTasks

Un esempio di relazione è presentata in tabella 6.58.

In tabella 6.59 è definito lo schema di relazione *ProcessingsOfSingleTasks*, usato per rappresentare la relativa classe del modello a oggetti.

L'attributo *ElemReqId* permette di definire l'associazione tra le classi 'Local Elementary Processings' e 'Processings Of Single Tasks', mentre per rappresentare l'associazione tra le classi 'Processings Of Single Tasks' e 'Tasks Of Productions' si considera l'attributo *TaskOfProdId*. Infine l'attributo *CompName* definisce la relazione tra le classi 'Processings Of Single Tasks' e 'Local Computers'.

In tabella 6.60 si presenta un esempio di relazione.

ElemReqId	TaskOfProdId	CompNam	Status	StartTime	EndTime
elp00001	prod1.t1	osde00	failed	20-01-1998 23:52	
elp00001	prod1.t1	osde00	succesfully completed	21-01-1998 08:02	21-01-1998 12:53
elp00001	prod1.t2	pccms1	succesfully completed	21-01-1998 13:01	21-01-1998 13:23
elp00001	prod1.t3	hpcms01	succesfully completed	21-01-1998 12:59	21-01-1998 19:23
elp00003	prod2.t1	pccms1	in execution	22-01-1998 11:50	
...	...	...	...	...	...

Tabella 6.60: Esempio di relazione

Capitolo 7

Architettura software del sistema

In questo capitolo viene descritta l'architettura software del sistema, definendo le caratteristiche e le funzionalità dei vari moduli, e le interazioni che esistono tra essi. Vengono presentati solo i moduli e le procedure più importanti, mentre non vengono trattati dettagli implementativi poco significativi, o procedure per le quali non si ritiene opportuna un'approfondita analisi (ad esempio le operazioni sui database).

Informazioni più dettagliate si possono comunque trovare in [1], nel documento in cui si descrive l'interfaccia delle varie classi.

7.1 Struttura del sistema

In figura 7.1 è rappresentata l'architettura del sistema DEMI, ad un primo livello di astrazione.

A livello globale, quindi, il sistema DEMI si compone di due moduli fondamentali: il **Global Interface** e il **DEMI Global Control System**. Il Global Interface viene utilizzato dai global administrator per interagire con il sistema: i comandi specificati vengono elaborati dal DEMI Global Control System, che "torna" poi all'amministratore i corrispondenti risultati.

Il DEMI Global Control System, nel corso della sua attività, accede ai database globali, sia per operazioni di lettura, che per operazioni di scrittura.

Per quel che riguarda l'architettura del sistema a livello locale, invece, si devono considerare altri due moduli: il **Local Interface** e il **DEMI Local Control System**. Gli utenti del sito locale (local administrator compresi) interagiscono con il sistema attraverso il Local Interface, e i comandi specificati vengono elaborati dal DEMI Local Control System, che può accedere sia ai database locali che ai database globali.

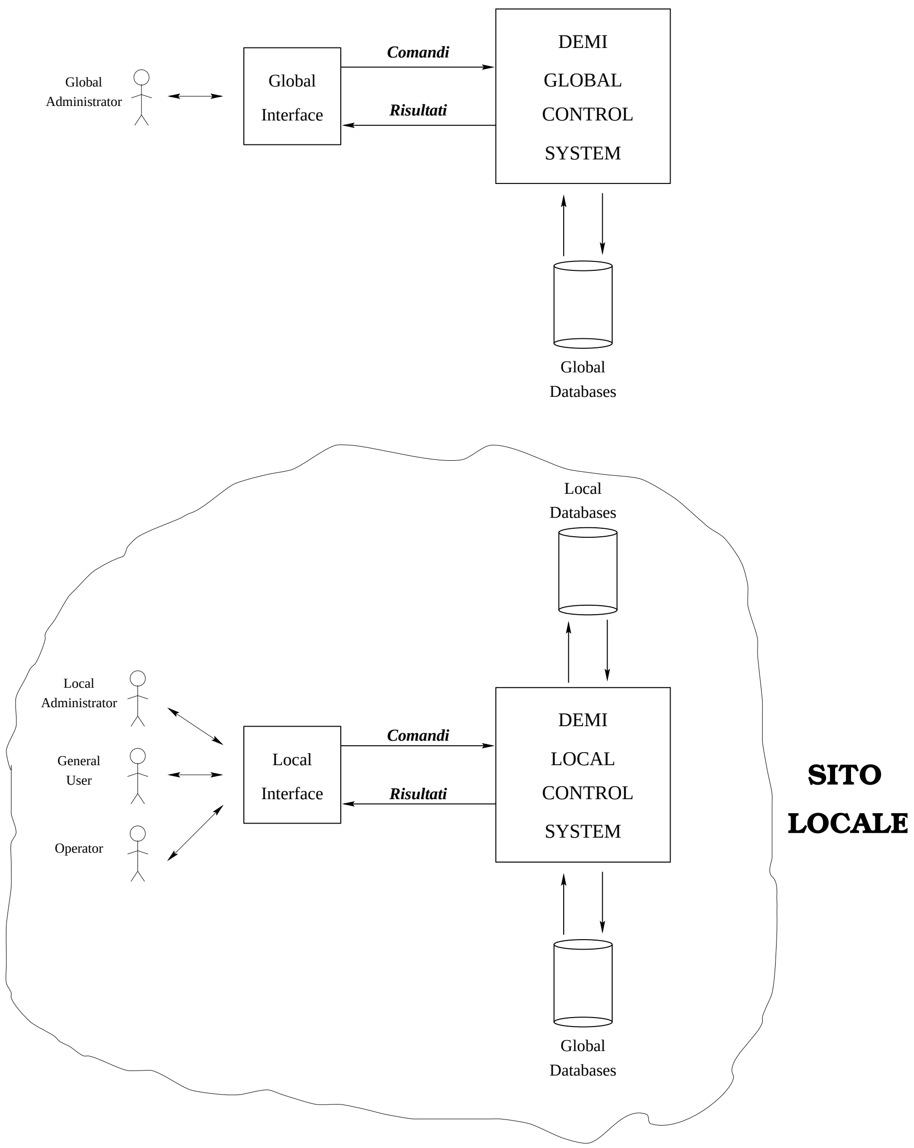


Figura 7.1: Architettura del sistema DEMI

Nei prossimi paragrafi vengono descritte le funzionalità di questi moduli.

7.2 Global Interface

Le operazioni che un global administrator può effettuare tramite il Global Interface sono:

- la modifica della propria password;
- la gestione del Global Administrator Database (definizione di nuovi amministratori, cancellazione di amministratori, operazioni di modifica, ricerca, visualizzazione);
- la gestione dell'Architecture Database (inserimento di nuovi elementi, cancellazione di architetture, modifiche, ricerche, visualizzazioni);
- la gestione del Tape Device Model Database (operazioni di inserimento, cancellazione, modifica, ricerca, visualizzazione);
- operazioni di ricerca e visualizzazione sul Data Database;
- operazioni di ricerca e visualizzazione sul Processing Database.

7.3 DEMI Global Control System

Il DEMI Global Control System elabora i comandi specificati dal global administrator. Si tratta di operazioni di inserimento, cancellazione, modifica, ricerca e visualizzazione effettuate sui database globali: per la descrizione di queste procedure si rimanda ai documenti presenti in [1].

Quando sarà realizzata la versione finale del sistema, nel DEMI Global Control System sarà implementato anche il Global Scheduler, che dovrà gestire lo scheduling delle varie elementary request, considerando un ambiente di calcolo distribuito geograficamente tra più siti.

7.4 Local Interface

Le operazioni che un utente può effettuare attraverso il Local Interface dipendono dal suo "ruolo" (un utente può essere un local administrator, e/o un utente generico, e/o un operatore).
Un local administrator può:

- modificare la sua password;
- gestire lo User Database (per operazioni di inserimento, cancellazione, modifica, ricerca, visualizzazione);
- effettuare operazioni di ricerca e visualizzazione sull'Architecture Database;
- effettuare operazioni di ricerca e visualizzazione sul Tape Device Model Database;
- gestire il Local Computer Database (per definire nuovi computer, per eliminare dal database calcolatori, per disabilitare e riabilitare computer, per modificare informazioni relative ad elementi definiti nel database, per operazioni di ricerca e visualizzazione);
- gestire il Local Buffer Database (per operazioni di inserimento, cancellazione, modifica, ricerca, visualizzazione);
- gestire il Local Tape Device Database (per operazioni di inserimento, cancellazione, modifica, ricerca, visualizzazione).

Un utente generico può:

- modificare la sua password;
- operare sui tipi di task (per definirne di nuovi, per specificare nuove implementazioni per determinate architetture, per operazioni di cancellazione, modifica, ricerca, visualizzazione);
- operare sulle production (per definirne di nuove, per modificarle, per cancellarle);
- operare sulle computational request (per sottomettere nuove richieste al sistema, per verificare lo stato di esecuzione di una richiesta, per "abortire" l'esecuzione di una richiesta);
- eseguire operazioni di ricerca e visualizzazione:
  - sull'Architecture Database;
  - sul Tape Device Model Database;
  - sul Local Computer Database;
  - sul Local Buffer Database;

  - sul Local Tape Device Database;
  - sul Data Database;
  - sul Processing Database.

Infine un operatore può semplicemente:

- modificare la sua password;
- visualizzare lo stato dei tape device definiti nel Local Tape Device Database.

7.5 DEMI Local Control System

Il DEMI Local Control System gestisce i comandi specificati dagli utenti attraverso il Local Interface.

Non si esaminano qui tutte le funzionalità di questo modulo (per un'analisi completa si rimanda ai documenti presenti in [1]): in questo paragrafo e nel successivo viene considerata solamente la parte fondamentale del DEMI Local Control System, ovvero l'elaborazione delle richieste computazionali sottomesse dagli utenti.

In questo paragrafo vengono descritti, ad alto livello, i vari moduli del DEMI Local Control System, che vengono esaminati più in dettaglio nel paragrafo 7.6.

La parte del DEMI Local Control System che gestisce l'elaborazione delle richieste computazionali è rappresentata in figura 7.2.

Innanzittutto, quindi, ogni richiesta computazionale sottomessa dagli utenti viene scomposta in elementary request, da un particolare modulo chiamato **Request Decomposer**.

Si consideri, ad esempio, una richiesta computazionale che fa riferimento alla produzione *prod1* rappresentata in figura 7.3 (*inp1*, *inp2*, *inp3* e *inp6* sono data box di input, *out7*, *out8* e *out9* sono data box di output, *dat4* e *dat5* sono data box che rappresentano dati temporanei).

Agli input di tipo singleton *inp1*, *inp3* e *inp6* siano associati rispettivamente i singleton input file *sinfile1*, *sinfile3* e *sinfile6*, mentre all'input di tipo stream *inp2* sia associato lo stream input set rappresentato dalla lista di file *file1*, *file2*, ..., *filen*, che devono quindi essere elaborati. Il Request Decomposer scompone questa richiesta in n elementary request: *er-1*, *er-2*, ... *er-n*, come mostrato in figura 7.4. I file associati agli input *inp1*, *inp2*, *inp3* e *inp6* per queste n elementary request sono rispettivamente (*sinfile1*, *file1*, *sinfile3*, *sinfile6*), (*sinfile1*, *file2*, *sinfile3*, *sinfile6*), ..., (*sinfile1*, *filen*, *sinfile3*, *sinfile6*).

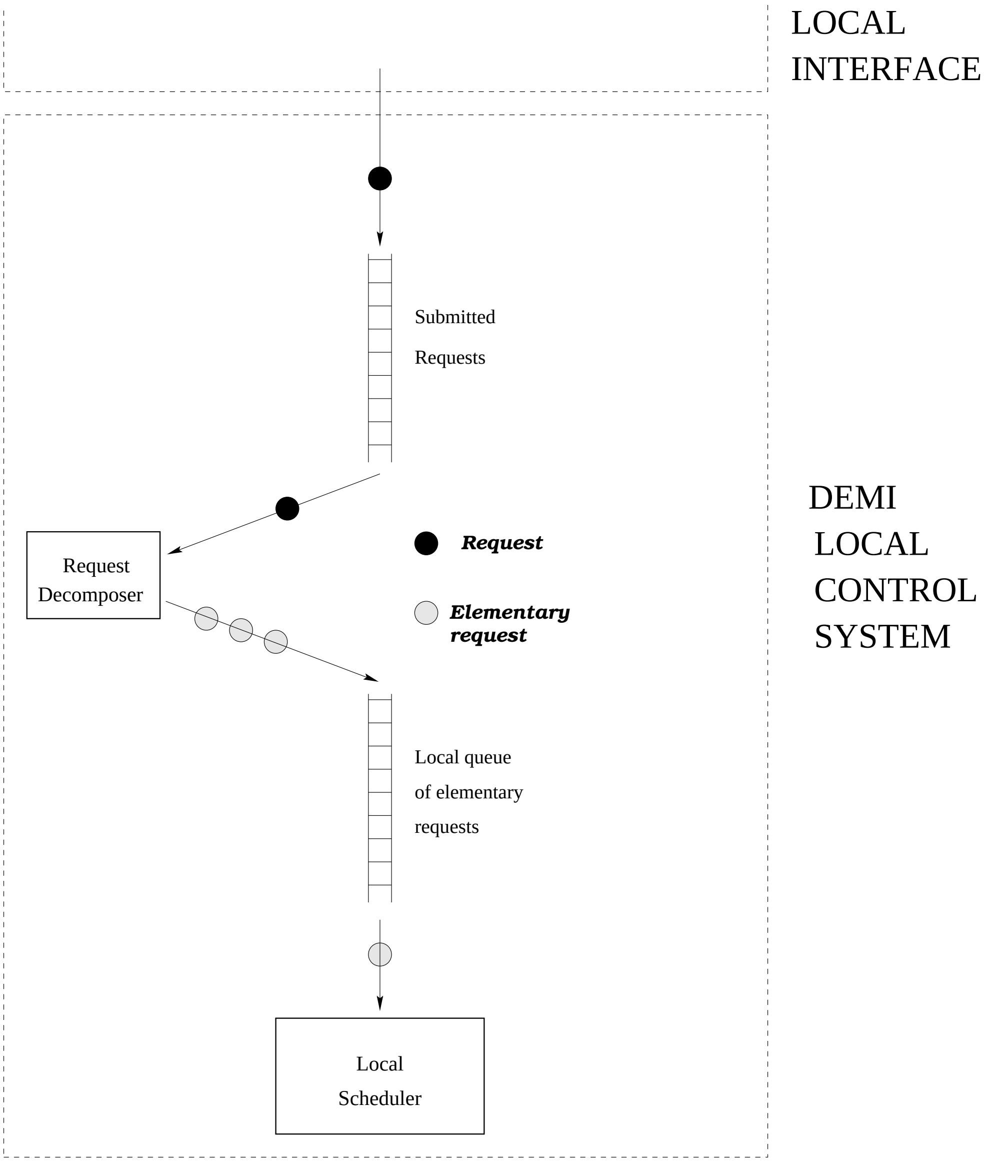


Figura 7.2: La gestione delle elaborazioni delle richieste computazionali

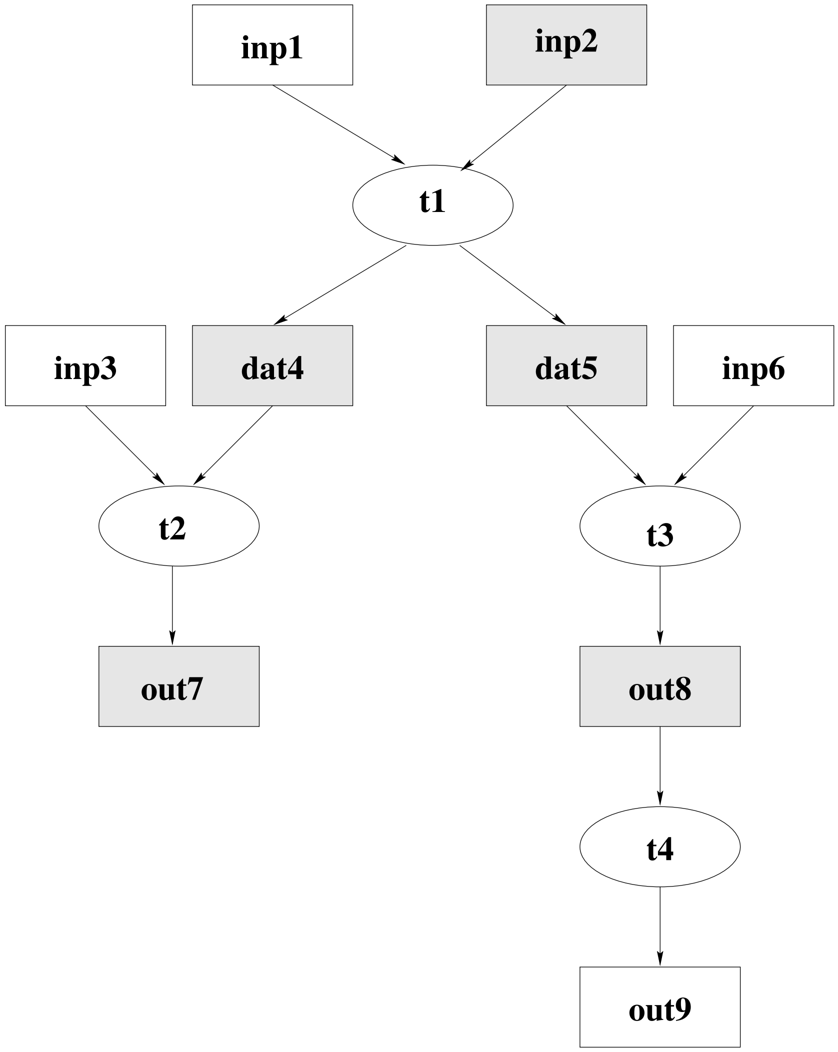


Figura 7.3: Produzione *prod1*

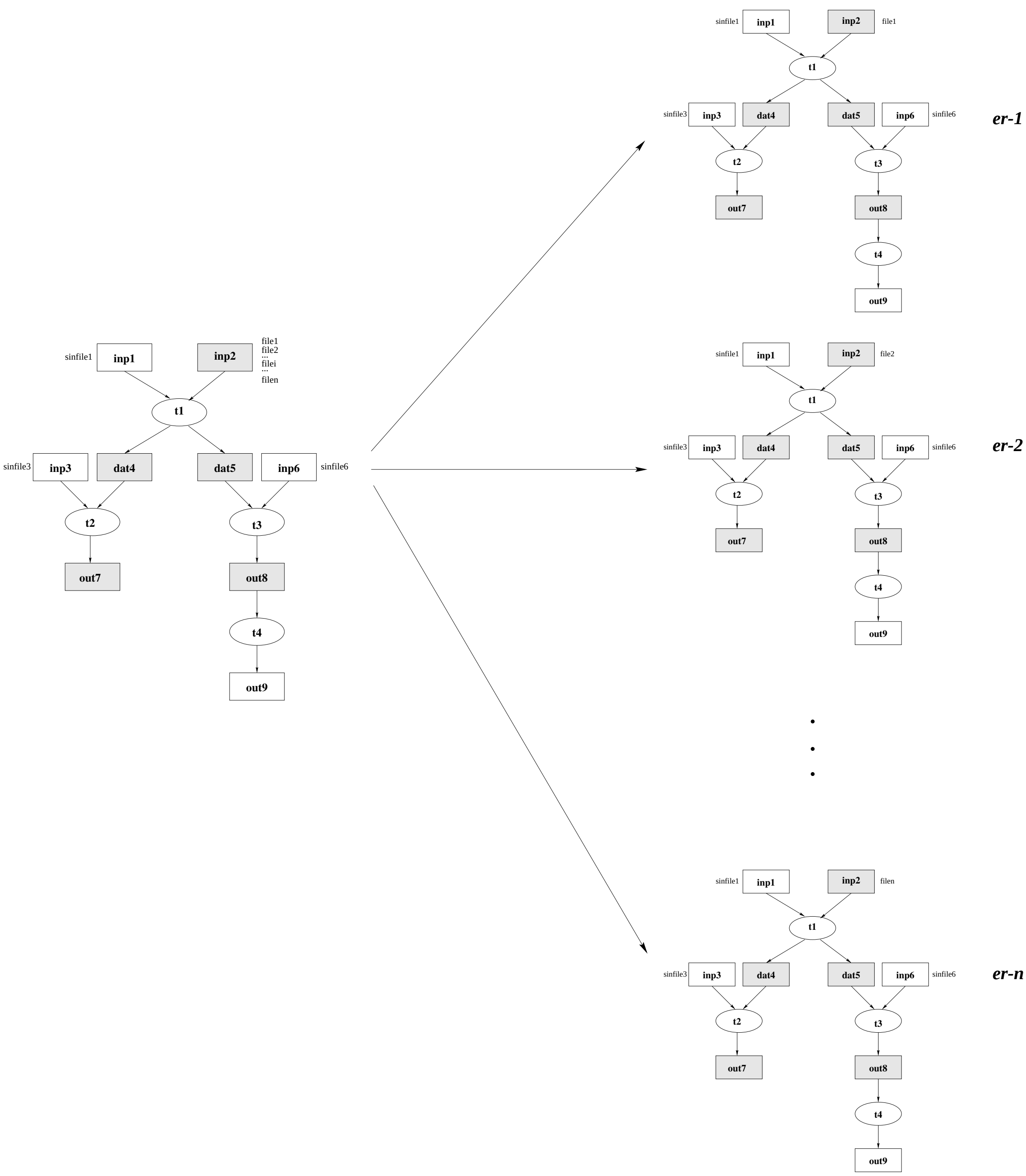


Figura 7.4: Scomposizione di una richiesta in elementary request, da parte del Request Decomposer

Se non ci fossero input di tipo stream, ogni elementary request corrisponderebbe ad un'iterazione (il numero di iterazioni deve essere stato specificato dall'utente che ha sottomesso la richiesta).

Le elementary request vengono inserite nella 'local queue of elementary requests', dove vengono poi esaminate, una alla volta (in base alla loro priorità), dal Local Scheduler.

Il **Local Scheduler** è il modulo fondamentale del DEMI Local Control System: questa "entità" è infatti responsabile della gestione delle elaborazioni delle elementary request, utilizzando le risorse di calcolo disponibili localmente.

L'architettura del Local Scheduler può essere schematizzata dal diagramma di figura 7.5.

Per ogni elementary request il Local Scheduler, prima di tutto, verifica se essa può essere effettivamente elaborata nel sito locale. L'elementary request viene poi esaminata da un particolare modulo, chiamato **Elementary Request Decomposer**, che la scompone nei task che costituiscono la corrispondente produzione.

Con riferimento all'esempio prima considerato, l'elementary request *er-i* viene scomposta in quattro task, *er-i-t1*, *er-i-t2*, *er-i-t3* e *er-i-t4*, associati rispettivamente ai task *t1*, *t2*, *t3* e *t4* della produzione *prod1*, come mostrato in figura 7.6.

Quindi, mentre il Request Decomposer scompone una richiesta in elementary request (ogni elementary request corrisponde ad un singolo stream input file dello stream input set o, se la corrispondente produzione non ha input di tipo stream, ad un'iterazione), l'Elementary Request Decomposer scompone ogni elementary request nei task che compongono la corrispondente produzione (se la produzione è costituita da n task, combinati opportunamente tra loro, l'elementary request viene scomposta in n task).

Il modulo **Task Starter** si occupa di verificare quali sono i task pronti per essere eseguiti (i task per cui sono disponibili tutti i relativi input).

Ad esempio, per l'elementary request *er-i*, il Task Starter rileva che il task *er-i-t1* può essere eseguito: è infatti l'unico task dell'elementary request *er-i* per cui sono disponibili tutti i relativi file di input (*sinfile1* e *filei*). Si sceglie un'architettura su cui eseguire il task, in base a vari parametri (le performance dei computer, il parametro 'cost of the task', i job già assegnati ai vari computer), e si sottomette il task al batch system, ovvero ad un sistema di load sharing in grado di gestire l'esecuzione di batch job, indicando che l'esecuzione deve avvenire su uno dei computer dell'architettura scelta. Prima però viene allocato sui disk buffer lo spazio per i file di output che saranno prodotti nell'esecuzione di questo task (nell'esempio considerato, quando il task *er-i-t1* deve essere eseguito, viene allocato lo spazio per la memorizzazione dei file associati ai data box *dat4* e *dat5*).

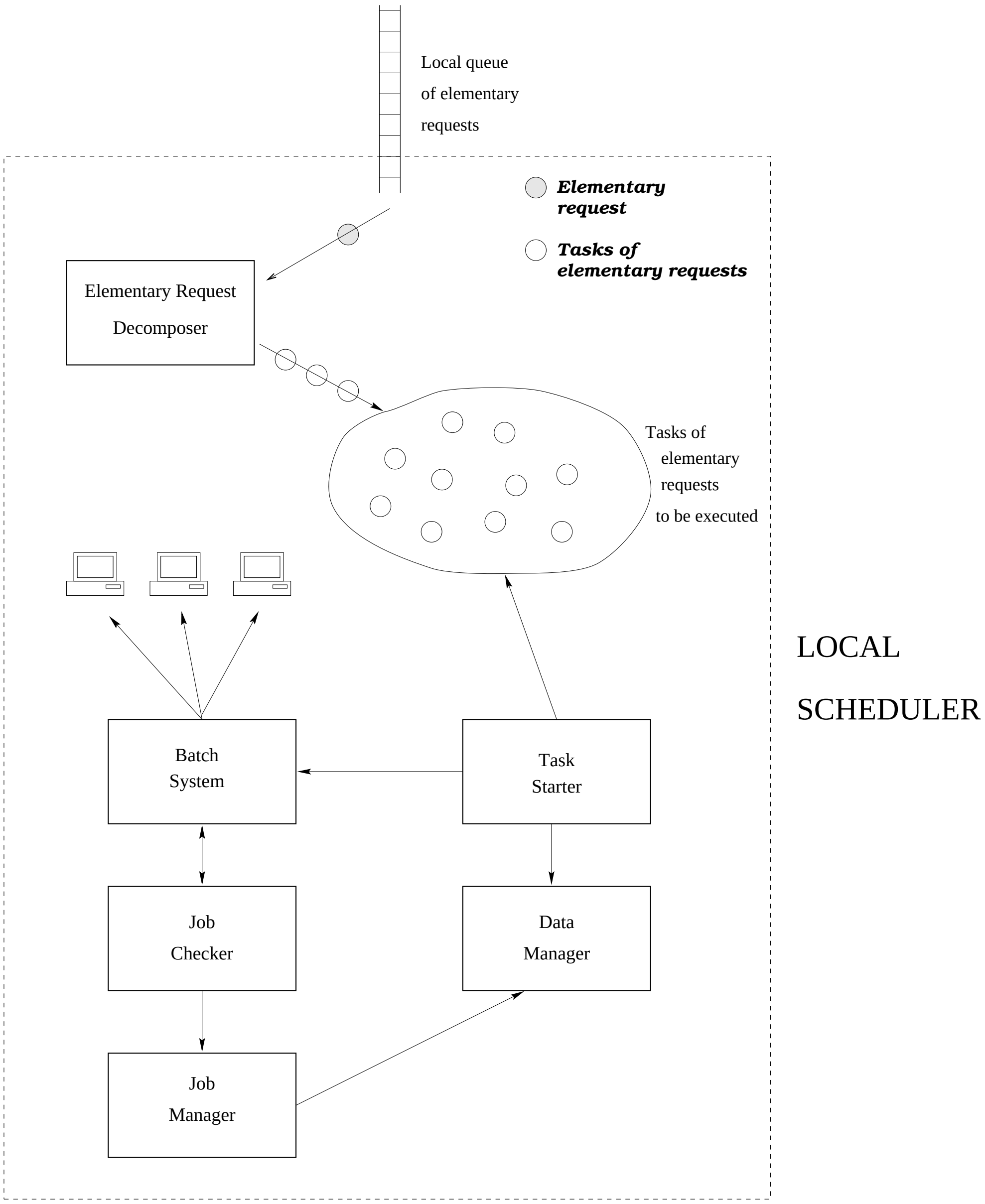


Figura 7.5: Architettura del Local Scheduler

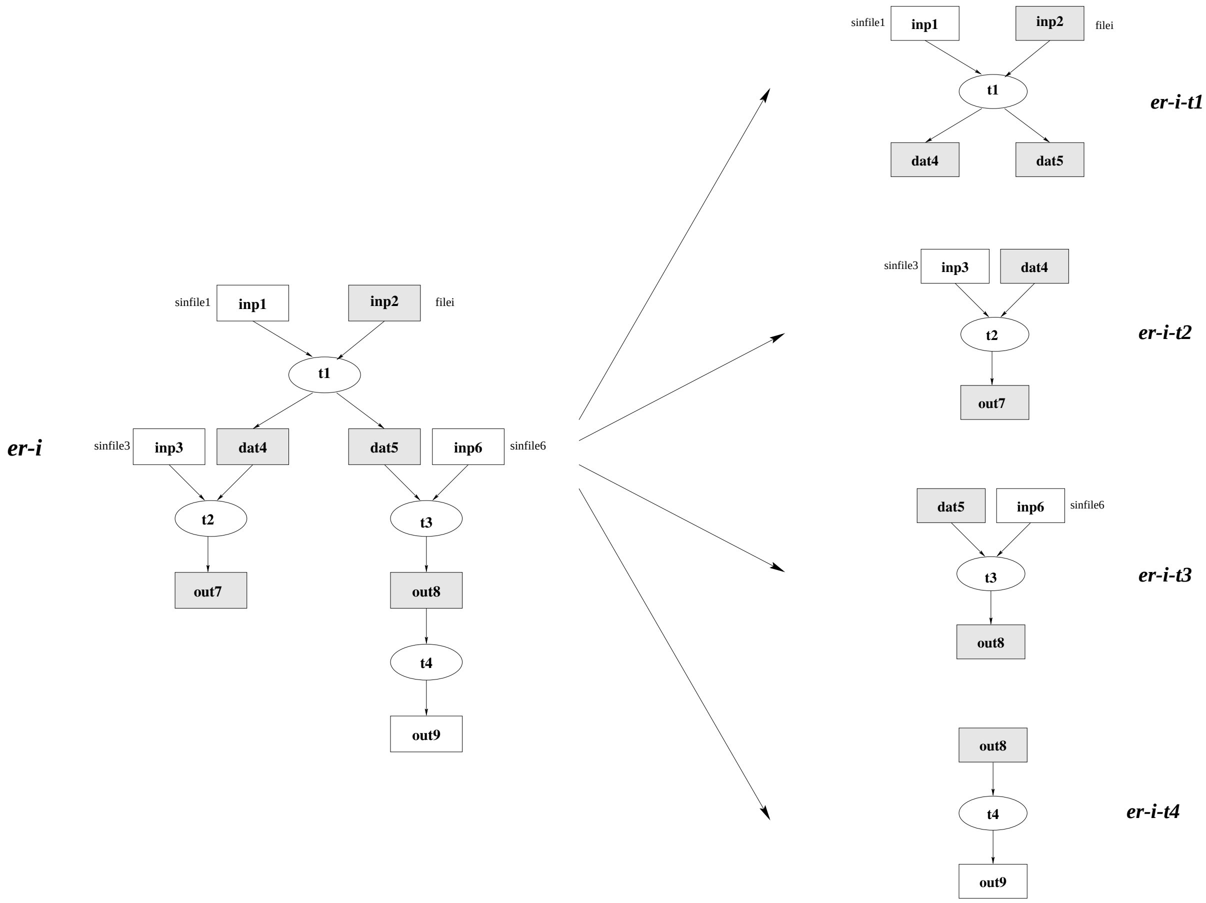


Figura 7.6: Scomposizione di una elementary request da parte dell'Elementary Request Decomposer

Ci si accerta che i file usati come input dal task (nel nostro caso i file *sinfile1* e *filei*) siano presenti su disco, ed eventualmente viene effettuata la copia da nastro.

È poi compito del batch system scegliere il computer che dovrà eseguire il task, in modo da ottimizzare l'uso complessivo delle varie risorse. Il batch system gestisce anche il suspend-resume dei job, secondo la configurazione considerata.

Un modulo chiamato **Job Checker** si "interfaccia" con il batch system, per verificare lo stato dei job DEMI. In particolare il Job Checker individua quando termina con successo l'esecuzione di un job, o quando l'esecuzione di un task fallisce. Il modulo **Job Manager** è responsabile nel gestire in maniera opportuna questi eventi.

Quando il Job Checker rileva che è terminata l'esecuzione di un task, si cancellano gli eventuali stream file "temporanei" non più necessari. Il Job Manager controlla poi se l'elaborazione dell'elementary request considerata è terminata, oppure se ci sono nuovi task da eseguire.

Sempre con riferimento all'elementary request *er-i* dell'esempio considerato, quando il task *er-i-t1* termina la sua esecuzione, il Task Starter sottomette al batch system per l'esecuzione i task *er-i-t2* e *er-i-t3*, per cui sono disponibili tutti i relativi file di input. Quando termina l'elaborazione del task *er-i-t2*, il file associato al data box *dat4* viene cancellato, perché non più necessario. Quando termina il processing del task *er-i-t3*, il file associato al data box *dat5* viene cancellato, e il Task Starter invoca l'esecuzione del task *er-i-t4*. Quando anche questo task termina la sua esecuzione, il processing dell'elementary request *er-i* è terminato, in quanto tutti i suoi task sono stati elaborati.

Quando l'elaborazione di un'elementary request è terminata, gli stream output file (quindi, nel nostro caso, i due file associati rispettivamente ai data box *out7* e *out8*), vengono registrati nel Data Database. Questi file di output vengono poi "spostati" dove richiesto dall'utente che ha sottomesso la richiesta computazionale (su nastro, su una directory privata, o mantenuti sui disk buffer).

Quando poi termina il processing di una richiesta, ovvero quando sono state elaborate tutte le sue elementary request, si analizzano i singleton file che sono stati prodotti: i file temporanei vengono cancellati, mentre i file di output vengono registrati nel Data Database, e spostati nel posto richiesto dall'utente (nell'esempio considerato l'unico singleton input file da considerare è quello associato al data box *out9*).

Se invece il Job Checker rileva che l'esecuzione di un task non è terminata con successo (ad esempio a causa di un crash del computer dove il job era in esecuzione), si invoca il Task Starter, per fare in modo che questo task venga rieseguito.

La gestione delle operazioni sui file di dati è affidata ad uno specifico modulo,

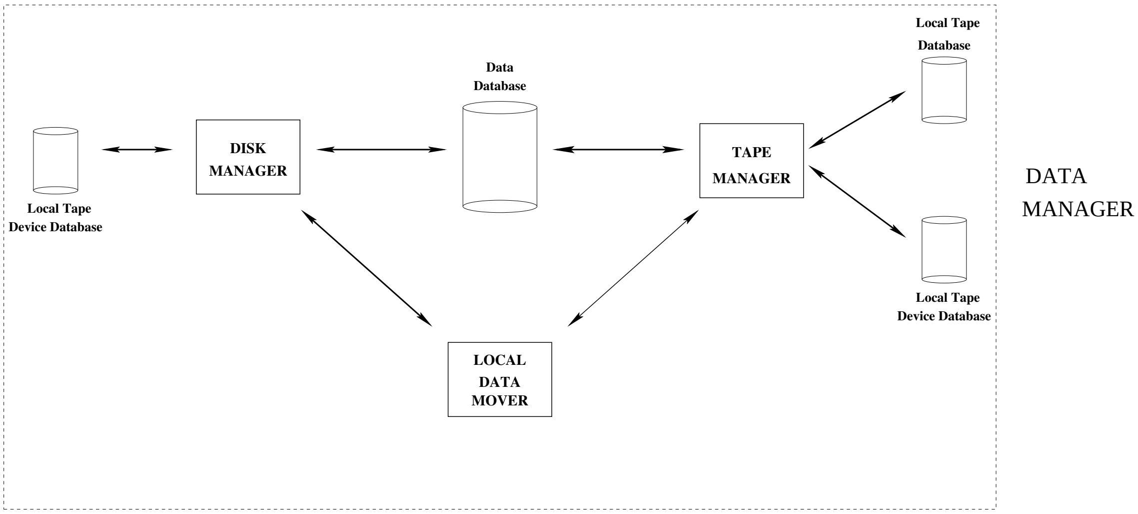


Figura 7.7: Architettura del Data Manager

chiamato **Data Manager**: le sue funzionalità vengono utilizzate sia dal Task Starter, che dal Job Manager.

L'architettura del Data Manager è rappresentata in figura 7.7.

Il Data Manager è costituito da tre moduli fondamentali: il **Disk Manager**, che gestisce le operazioni che riguardano i file memorizzati sui disk buffer, il **Tape Manager**, che si occupa delle procedure che coinvolgono i file di dati memorizzati su nastri, e il **Local Data Mover**, che gestisce le copie di file da disco a nastro, e viceversa.

7.6 Analisi del DEMI Local Control System

In questo paragrafo vengono esaminati più in dettaglio i moduli del DEMI Local Control System descritti nel paragrafo precedente, in particolare per analizzare le operazioni che essi devono gestire.

In questa descrizione si fa riferimento alle tabelle della base di dati relazione descritta nel capitolo 6, utilizzata per memorizzare in maniera persistente le varie informazioni.

7.6.1 Request Decomposer

Il seguente pseudo-codice descrive l'operazione di scomposizione delle richieste computazionali, effettuata dal Request Decomposer:

```
RequestDecomposer(req : Request);
begin
  prod = ProductionOfRequest(req);
  if (esiste inp input di tipo stream di prod)
    then for each f stream input file associato a inp
            do
            er = NewElementaryRequest();
            InsertIntoTable_StreamIOOfElementaryRequests(f);
            InsertIntoLocalQueue(er);  od
     else for i=1 to NumIt
            do
            er = NewElementaryRequest();
            InsertIntoLocalQueue(er);  od fi
end
```

Si esamina, quindi, se per la corrispondente production esiste un input di tipo stream (una produzione può avere al più un input di questo tipo). In caso positivo viene definita una elementary request per ogni stream input file associato a questo input. Se quindi per questo input si devono elaborare n file, vengono definiti nella tabella *ElementaryRequests* n elementi: ad ognuno di essi viene associato uno specifico identificatore (il campo 'ElemReqId'), e al campo 'Status' viene associato il valore 'Waiting to be processed'. Ognuno di questi stream input file viene definito nella tabella *StreamIOOfElementaryRequests*. Se invece la produzione non ha input di tipo stream, si definiscono un numero di elementary request pari al valore del parametro 'NumIt' della tabella *Requests* (il numero di iterazioni). In questo caso non viene definito nessun elemento nella tabella *StreamIOOfElementaryRequests*.

Le elementary request vengono poi inserite nella 'local queue of elementary requests', in modo che possano essere analizzate da parte del Local Scheduler.

Il Request Decomposer viene "attivato" ogni volta che un utente sottomette al sistema una nuova richiesta computazionale.

7.6.2 Local Scheduler

Il Local Scheduler "estrae" le elementary request dalla 'local queue of elementary requests' una alla volta: viene scelta l'elementary request con priorità più elevata (nel caso esistano più elementary request con stessa priorità, si sceglie quella che è stata inserita per prima nella coda).

Dopo aver verificato che l'elementary request può essere effettivamente elaborata nel sito locale (vedi par. 7.6.3), essa viene esaminata dall'Elementary Request Decomposer (vedi par. 7.6.4), che la scompone nei task che costituiscono la corrispondente produzione.

Il modulo Task Starter (vedi par. 7.6.5) si occupa di verificare quali sono i task pronti per essere eseguiti: per la loro esecuzione si utilizzano le funzionalità di un batch system.

Il Job Checker (vedi par. 7.6.7) si "interfaccia" con il batch system, per verificare lo stato dei job DEMI. In particolare individua quando inizia l'esecuzione di un task prima "sottomesso", quando termina con successo l'esecuzione di un job, quando l'esecuzione di un task fallisce. Il Job Manager è responsabile nel gestire in maniera opportuna questi eventi (vedi par. 7.6.8).

7.6.3 Controlli preliminari sull'elementary request

La procedura che deve essere eseguita prima che un'elementary request venga scomposta dall'Elementary Request Decomposer può essere rappresentata dal flow chart di figura 7.8.

Si eseguono quindi dei controlli preliminari, per verificare che sia effettivamente possibile elaborare l'elementary request nel sito locale.
Si devono fare tre tipi di controlli:

1. verificare che i file di input della elementary request siano presenti nel sito locale;

2. verificare che nel sito siano presenti computer di tutte le architetture necessarie per l'esecuzione dei vari task;

3. verificare che nel sito siano presenti i tape device necessari per l'eventuale lettura/scrittura da/su nastro dei file di input/output di questa elementary request.

Se i test non hanno successo (e quindi non è possibile elaborare l'elementary request localmente):

- si avverte del problema l'utente che ha sottomesso la richiesta;

- si modifica l'attributo 'Status' della tabella *ElementaryRequests*, per l'elemento che rappresenta l'elementary request considerata, ponendo questo campo pari a 'Impossible to process';

- si controlla se si è terminato di elaborare la richiesta associata a questa elementary request e, in caso positivo, si gestisce opportunamente la situazione (vedi par. 7.6.10).

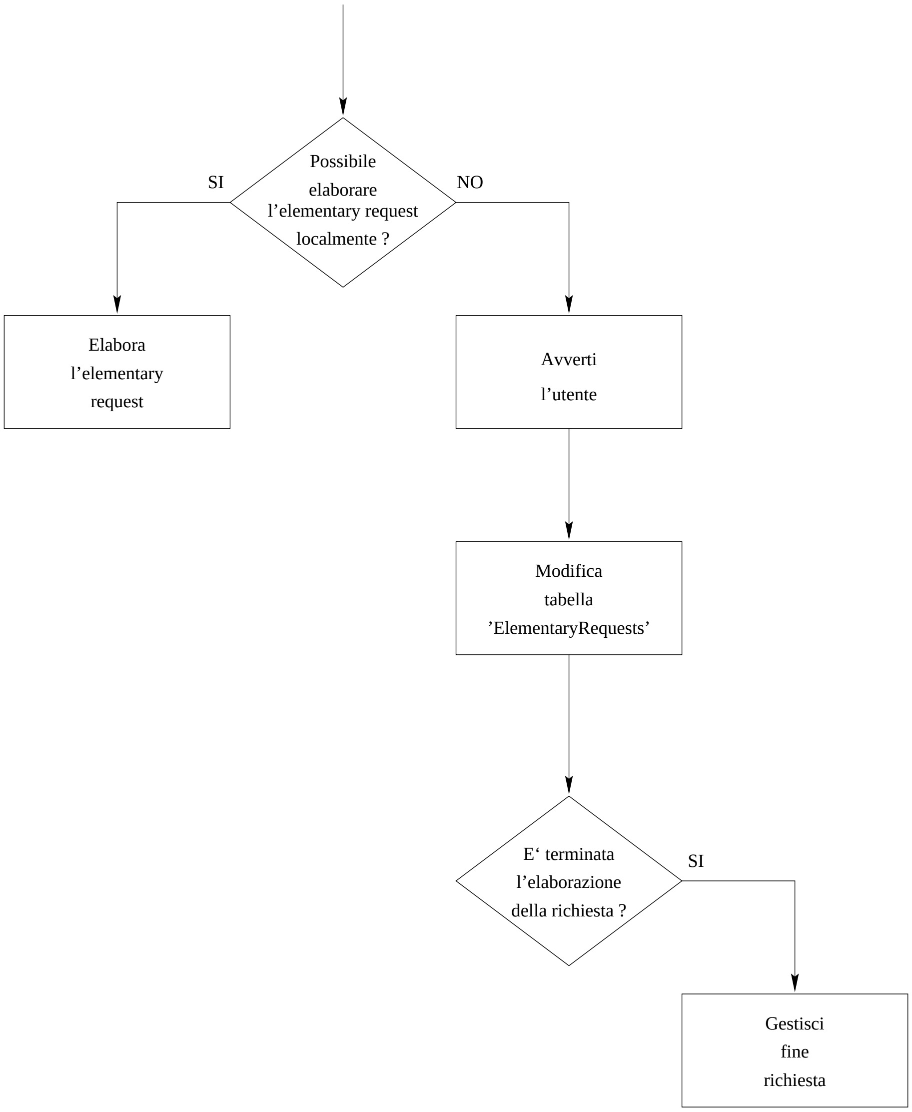


Figura 7.8: Analisi dell'elementary request

ElemReqId	TaskOfProdId	Status	JID
er-i	prod1.t1	succesfully processed	12413
er-i	prod1.t2	waiting inputs	
er-i	prod1.t3	waiting inputs	
er-i	prod1.t4	waiting inputs	
er-x	prod3.tx	succesfully processed	12432
er-x	prod3.ty	processing	12435
...	...	...	...

Tabella 7.1: Tabella 'TasksOfElementaryRequests'

Se invece i controlli preliminari hanno avuto esito positivo, l'elementary request è pronta per essere elaborata dall'Elementary Request Decomposer (vedi par. 7.6.4).

7.6.4 Elementary Request Decomposer

Il modulo Elementary Request Decomposer scompone ogni elementary request nei task che costituiscono la corrispondente produzione.

Prima di questa operazione, viene modificato il campo 'Status' della tabella *ElementaryRequests* per l'elemento considerato: viene definito il nuovo valore 'Processing', per indicare appunto che inizia l'elaborazione.

Per tener traccia dello stato dell'elaborazione dei vari task delle varie elementary request, si utilizza una particolare struttura dati, gestita in maniera persistente, chiamata *TasksOfElementaryRequests*, rappresentata in tabella 7.1.

Ogni elemento è rappresentato da una quaterna di valori: l'identificatore della elementary request, l'identificatore del task componente la corrispondente produzione, lo stato dell'elaborazione di questo task, e un Job Identifier (JID) in seguito descritto. Ogni task "generato" dall'Elementary Request Decomposer viene definito in questa tabella: vengono inizializzati i campi 'ElemReqId', 'TaskOfProdId' e 'Status' (con il valore 'Waiting inputs', per indicare che il task, per poter iniziare l'esecuzione, è in attesa dei relativi input), mentre per il momento all'attributo 'JID' non viene assegnato nessun valore. Considerando l'elementary request *er-i*, dell'esempio descritto nel paragrafo 7.5, nella tabella 'TasksOfElementaryRequests' vengono definiti quattro elementi, relativi ai quattro task della production *prod1*.

Quando si elabora una elementary request, è necessario conoscere anche quali sono i file associati ai vari data box della corrispondente produzione: questi file vengono infatti usati come input-output dai vari task. A questo scopo si utilizza

ElemReqId	DataBoxId	FileId
er-i	prod1.inp1	sinfile1
er-i	prod1.inp3	sinfile3
er-i	prod1.inp6	sinfile6
er-i	prod1.inp2	filei
er-i	prod1.dat4	filei-4
er-i	prod1.dat5	filei-5
er-x	prod3.ds1	filey
...	...	...

Tabella 7.2: Tabella 'FilesOfElementaryRequests'

un'altra struttura, chiamata *FilesOfElementaryRequests*, rappresentata in tabella 7.2.

Ogni elemento di questa tabella viene rappresentato da una terna: l'identificatore dell'elementary request, l'identificatore del data box della produzione, e l'identificatore del file. L'Elementary Request Decomposer, esaminando le tabelle *StreamIOOfElementaryRequests* e *SingletonIOOfRequests*, registra nella tabella *FilesOfElementaryRequests* gli elementi relativi agli input file (di tipo stream e di tipo singleton) dell'elementary request considerata. Sempre con riferimento all'elementary request *er-i* dell'esempio considerato, vengono definiti nella tabella 'FilesOfElementaryRequests' i file *sinfile1*, *sinfile2*, *sinfile6* e *filei*, associati rispettivamente ai data box *inp1*, *inp3*, *inp6* e *inp2*.

Dopo queste operazioni, viene "attivato" il Task Starter, per verificare quali task di questa elementary request sono pronti per essere eseguiti: a tal proposito si veda il paragrafo 7.6.5.

Queste procedure sono descritte nel seguente pseudo-codice:

```
ElementaryRequestDecomposer(er : ElementaryRequest);
begin
  ModifyTable_ElementaryRequests();
  req = RequestOfElementaryRequest(er);
  prod = ProductionOfRequest(req);
  for each t task di prod
      do InsertIntoTable_TasksOfElementaryRequests(t); od
  for each f1 singleton input file di req
      do InsertIntoTable_FilesOfElementaryRequests(f1); od
  for each f2 stream input file di er
      do InsertIntoTable_FilesOfElementaryRequests(f2); od
  TaskStarter(er);
end
```

L'Elementary Request Decomposer viene implementato come un processo daemon, che viene "attivato" quando sono presenti elementi nella 'local queue of elementary requests'.

7.6.5 Task Starter

Il Task Starter è il modulo che deve verificare quali sono i task pronti per essere eseguiti (i task per i quali i relativi input sono disponibili), e ne deve gestire l'esecuzione.

Per controllare quali sono i task di una elementary request che possono essere eseguiti, il Task Starter analizza la tabella *TasksOfElementaryRequests*, considerando tutti i task relativi all'elementary request considerata per cui l'attributo 'Status' sia pari a 'Waiting inputs' (i task, quindi, che devono ancora essere eseguiti, perché in attesa dei corrispondenti file di input). Per ognuno di questi task si controlla, esaminando la tabella *FilesOfElementaryRequests*, se tutti i relativi file di input sono disponibili. In caso positivo il task può essere eseguito.

Si consideri, ad esempio, l'elementary request *er-i*. Il contenuto delle tabelle *TasksOfElementaryRequests* e *FilesOfElementaryRequests* sia rispettivamente quello delle tabelle 7.1 e 7.2. In questo caso il Task Starter rileva che i task *t2* e *t3* sono pronti per essere eseguiti, in quanto per tutti i relativi data box di input della produzione (*inp3* e *dat4*, *dat5* e *inp6*) sono definiti i corrispondenti file.

Il seguente codice descrive sommariamente le operazioni che devono essere effettuate quando il Task Starter rileva che un task può essere eseguito:

```
TaskStarter(t : Task; er : ElementaryRequest);
begin
  ModifyTable_TasksOfElementaryRequests();
  for each f input file di t
      do
      if (necessary)
        then CopyFromTapeToBuffer(f); fi
      Lock(f); od
  for each outst stream output di t
      do
      siz = GetExpectedSize(outst);
      f1 = NewFile(siz); od
  for each outsin singleton output di t
      do
      if (necessary)
        then
              siz = GetExpectedSize(outsin);
              f2 = NewFile(siz); fi od
  tt = TaskType(t);
  arch = ChooseArchitecture(tt);
  Submit(t, arch);
end
```

Per prima cosa si modifica l'attributo 'Status' del corrispondente elemento della tabella *TasksOfElementaryRequests*, con il nuovo valore 'Processing'.

È poi necessario accertarsi che tutti i file di input del task siano presenti su disco (si controlla la tabella *FilesOnBuffers*). Se un file non è presente su un disk buffer, lo si copia da nastro a disco: questa operazione viene gestita dal Data Manager, come descritto nel paragrafo 7.6.11.

Per tutti i file di input del task si definisce uno stato di "lock", per evitare che questi vengano spostati altrove durante l'elaborazione.

Per ogni output di tipo stream del task, si analizza il corrispondente elemento della tabella *DataBoxesOfRequests*, per verificare qual è la massima dimensione che deve essere allocata per il file che verrà generato. Attraverso il Data Manager si definisce un nuovo elemento nel Data Database per questo file, allocando lo spazio necessario.

Si esaminano poi gli output di tipo singleton del task. Per ognuno di essi si controlla se il corrispondente file esiste già (potrebbe essere già stato creato da un'altra elementary request della stessa richiesta computazionale). In caso negativo è necessario definire il relativo file di output nel Data Database: la procedura è del tutto analoga a quella vista per gli output di tipo stream.

Si individua qual è il tipo del task considerato, che può essere stato implementato per diverse architetture: se ne deve scegliere una per l'esecuzione (questa procedura è descritta nel paragrafo 7.6.6).

Dopo queste operazioni il task è pronto per essere "sottomesso" al batch system per la relativa esecuzione, che dovrà essere eseguita su un computer dell'architettura selezionata.

Si "costruisce" la riga di comando che deve essere specificata per l'esecuzione del task, utilizzando le informazioni memorizzate nelle tabelle *ImplementationsOfTasks* e *ParametersForTasks*. Per quel che riguarda i parametri che rappresentano gli input-output, il pathname di questi file viene trovato esaminando la tabella *FilesOnBuffers*.

Analizzando la tabella *LocalComputers*, si verifica quali sono i computer dell'architettura prima selezionata presenti nel sito (si considerano solo i calcolatori il cui stato è pari a 'Enabled', quindi i computer che possono "accettare" task da eseguire). Utilizzando le apposite API fornite dal batch system, si sottomette al sistema la riga di comando che invoca l'esecuzione del task, indicando che l'elaborazione deve avvenire su uno di questi computer.

È compito del batch system decidere su quale computer eseguire l'elaborazione. Il batch system gestisce anche il suspend/resume dei job, secondo le regole specificate dall'amministratore del pool di workstation (vedi par. 5.1).

La routine usata per il submit del task "torna" un identificatore, chiamato JID (Job IDentifier). Utilizzando questo valore, si definisce l'attributo 'JID' della tabella *TasksOfElementaryRequests* per l'elemento che rappresenta questo task.

Infine si crea un elemento nella tabella *ProcessingsOfSingleTasks*, per definire l'elaborazione del task considerato: all'attributo 'Status' viene associato il valore 'Submitted', per indicare che il task è stato sottomesso al batch system per la relativa esecuzione.

7.6.6 Scelta di un'architettura per l'esecuzione di un task

L'algoritmo per scegliere su quale architettura eseguire un task può essere facilmente modificato (senza dover cambiare anche il codice di tutti gli altri moduli), in modo da definire nuovi parametri, nuovi criteri, ecc...

Al momento si utilizza un algoritmo molto semplice, in cui si considera il numero dei computer presenti per le varie architetture, le loro performance, il costo per l'esecuzione del task sulle varie architetture, i job già assegnati ai vari computer.

Siano A_1, A_2, ..., A_n le architetture per cui esiste un'implementazione del task considerato. Siano $C_{i,1}$, $C_{i,2}$, ..., $C_{i,Ni}$ i computer dell'architettura A_i abilitati ad elaborare task, e siano $P_{i,1}$, $P_{i,2}$, ..., $P_{i,Ni}$ le performance di questi computer (si considera l'attributo 'Performances' del Local Computer Database: vedi par. 4.3.1). Sia $COST_i$ il costo per l'esecuzione del task considerato sull'architettura A_i (il valore del parametro 'cost of the task' del Task Database: vedi par. 4.5.1). Sia n_i il numero di task assegnati ai computer dell'architettura A_i per l'esecuzione, la cui elaborazione deve ancora essere completata (task in quel momento già in esecuzione, o in attesa di essere eseguiti). Per l'esecuzione del task si sceglie l'architettura A_i per cui è massimo il rapporto:

$$\frac{\sum_{j=1}^{Ni} P_{i,j}}{COST_i + n_i}$$

7.6.7 Job Checker

Il **Job Checker** è il modulo del sistema DEMI che si occupa di rilevare lo stato dei job sottomessi al batch system.

In particolare viene rilevato l'inizio dell'esecuzione di un job (che passa dallo stato di suspend allo stato di run), l'elaborazione di un job completata con successo, e l'elaborazione fallita di un task. Questi eventi vengono gestiti dal Job Manager.

7.6.8 Job Manager

Inizio dell'esecuzione di un task

Quando si rileva che è iniziata l'esecuzione di un task su un certo calcolatore, si definiscono gli attributi 'CompName' e 'Status' (ponendo questo campo pari a 'Started') del corrispondente elemento della tabella *ProcessingsOfSingleTasks*.

Elaborazione di un task completata con successo

Il seguente pseudo-codice definisce le operazioni che devono essere eseguite dal Job Manager, quando è terminata l'elaborazione di un job di cui viene rilevato il corrispondente job identifier (JID):

EndTask(*JID* : *Integer*);
begin
  t = *TaskOfJID*(*JID*);
  er = *ElementaryRequestOfTask*(*t*);
  ModifyTable_TasksOfElementaryRequests();
  ModifyTable_ProcessingsOfSingleTasks();
  for *each f1 file generato da t*
    do *InsertIntoTable_FilesOfElementaryRequests*(*f1*); **od**
  for *each f2 stream file temporaneo usato come input da t e*
    non piu‘ necessario
    do *Delete*(*f2*); **od**
  if (*EndElReq*(*er*))
    then *ManageEndElReq*(*er*);
    else *TaskStarter*(*er*); **fi**
end

Analizzando la tabella *TasksOfElementaryRequests*, si identifica qual è il task che è terminato: per questo elemento della tabella *TasksOfElementaryRequests* si modifica il valore del campo 'Status', ponendolo pari a 'Succesfully processed'.

Si modifica anche il corrispondente elemento della tabella *ProcessingsOfSingle-Tasks*, definendo un nuovo valore per il campo 'Status' ('Succesfully processed'), e specificando la data e l'ora in cui l'elaborazione è terminata.

Si definiscono poi nella tabella *FilesOfElementaryRequests* gli output file (di tipo stream e di tipo singleton) prodotti nell'elaborazione di questo task. Gli stream file temporanei usati come input dal task, e non più necessari, vengono cancellati.

Si verifica poi se si è terminato di elaborare l'elementary request associata a questo task (si controlla il campo 'Status' della tabella *TasksOfElementaryRequests*, per vedere se esistono task di questa elementary request che devono ancora essere elaborati) e, in caso positivo, si gestisce in modo opportuno questa situazione (vedi par. 7.6.9). Se invece l'elaborazione dell'elementary request non è stata completata, si attiva il Task Starter per verificare se per essa ci sono nuovi task pronti per essere eseguiti (vedi par. 7.6.5): dovranno essere eseguiti quei task che usano come input i file di output del task appena terminato.

Elaborazione "fallita" di un task

Il codice sotto riportato descrive le operazioni che il Job Manager esegue quando il Job Checker rileva che l'esecuzione di un task, di cui viene rilevato il corrispondente Job Identifier, è fallita (non è stata completata con successo):

$CrashedTask(JID : Integer)$;
begin
  t = *TaskOfJID*(JID);
  ModifyTable_ProcessingsOfSingleTasks();
  NotifyUser();
  for *each f stream file prodotto da t*
    do
    Delete(f); **od**
  ModifyTable_TasksOfElementaryRequests();
  er = *ElementaryRequestOfTask*(t);
  TaskStarter(*er*);
end

Analizzando la tabella *TasksOfElementaryRequests* si identifica qual è il task associato al job identifier considerato. L'utente che ha sottomesso la corrispondente richiesta viene avvertito riguardo il problema. Si modifica poi il campo 'Status' dell'elemento della tabella *ProcessingsOfSingleTasks* relativo a questo task: viene definito il nuovo valore 'Crashed'. Si invoca poi il Data Manager, per eliminare gli stream file prodotti dal task. Si aggiorna il campo 'Status' dell'elemento della tabella *TasksOfElementaryRequests* corrispondente a questo task: il nuovo valore che viene inserito è 'Waiting inputs' (il task dovrà infatti essere rieseguito). Infine si invoca il Task Starter, per controllare se per l'elementary request corrispondente a questo task ci sono nuovi task pronti per essere eseguiti (vedi par. 7.6.5): con questa operazione si "forza" la riesecuzione del task considerato.

7.6.9 Elaborazione completata di una elementary request

Quando l'elaborazione di una elementary request è stata completata, per prima cosa si modifica il campo 'Status' del corrispondente elemento della tabella *ElementaryRequests*, definendo il nuovo valore 'Succesfully processed'.

Tutti gli output stream file che sono stati prodotti in questa elementary request devono essere "registrati". Questi file di output devono poi essere "spostati" dove specificato dall'utente che ha sottomesso la richiesta. Queste operazioni vengono gestite dal Data Manager (vedi par. 7.6.11).

I file di output devono anche essere definiti nella tabella *StreamIOOfElementaryRequests*, la struttura dati usata per memorizzare gli input-output di tipo stream delle elementary request.

Si cancellano poi tutti gli elementi delle tabelle *TaskOfElementaryRequests* e *FilesOfElementaryRequests* relativi all'elementary request considerata.

Si considerano gli stream input file utilizzati in questa elementary request: per ognuno di essi viene rimosso lo stato di “lock”, a meno che il file non sia stato “allocato” in un altro processing.

Infine si verifica se si è terminato di elaborare anche la corrispondente richiesta computazionale (esaminando la tabella *ElementaryRequests*, si controlla se esistono elementary request associate alla richiesta considerata per cui l’elaborazione è ancora in corso, o deve ancora avvenire), e in caso si gestisce opportunamente la situazione (vedi par. 7.6.10).

La seguente procedura descrive tutte queste operazioni:

ManageEndElReq(*er* : *ElementaryRequest*);
begin
  ModifyTable_ElementaryRequests();
  for *each f stream output file prodotto in er*
    do
    Register(*f*);
    MoveOutputFile(*f*);
    InsertIntoTable_StreamIOOfElementaryRequests(*f*); **od**
  ModifyTable_TasksOfElementaryRequests();
  ModifyTable_FilesOfElementaryRequests();
  for *each f stream input file usato come input in er*
    do *Unlock*(*f*); **od**
  req = *RequestOfElementaryRequest*(*er*);
  if (*EndReq*(*req*))
    then *ManageEndReq*(*req*); **fi**
end

7.6.10 Elaborazione completata di una richiesta

Quando è terminata l’elaborazione di una richiesta computazionale, per prima cosa si modifica il campo ’Status’ del corrispondente elemento della tabella *Requests*, ponendo questo attributo pari a ’Succesfully processed’.

Si analizzano poi i singleton output file che sono stati prodotti nell’elaborazione di questa richiesta: i file temporanei devono essere cancellati, mentre i file che corrispondono ad output della produzione devono essere “registrati”. È anche necessario spostare questi singleton output file dove indicato dall’utente che ha definito la richiesta: tutte queste procedure vengono gestite dal Data Manager, come descritto nel paragrafo 7.6.11.

I singleton output file devono anche essere definiti nella tabella *SingletonIOOfRequests*.

Si rimuove, poi, lo stato di “lock” associato a tutti i singleton input file utilizzati in questa richiesta, a meno che questi file non siano stati "allocati" anche in altri processing ancora in corso.

Infine si deve riportare il risultato dell'elaborazione al corrispondente utente: esaminando le tabelle *SingletonIOOfRequests* e *StreamIOOfElementaryRequests*, si trovano rispettivamente tutti i singleton output file associati alla richiesta considerata, e gli stream output file associati alle sue elementary request. La lista degli identificatori di tutti questi file viene inviata, via mail, all'utente che ha sottomesso la richiesta.

Il seguente algoritmo rappresenta in maniera schematica la procedura appena descritta:

```
ManageEndReq(req : Request);
begin
  ModifyTable_Requests();
  for each f singleton output file prodotto in req
     do
     if (f e' un file temporaneo)
       then Delete(f);
        else Register(f);
             MoveOutputFile(f);
             InsertIntoTable_SingletonIOOfRequests(f);  fi od
  for each f singleton input file usato come input in req
     do Unlock(f);  od
  NotifyResultsToUser();
end
```

7.6.11 Data Manager

Il Data Manager è il modulo che si occupa di gestire tutte le operazioni che riguardano i file di dati.
Come detto nei paragrafi precedenti, in particolare il Data Manager:

- copia i file di input dei task da nastro a disco;
- definisce nel Data Database i nuovi file di dati;
- registra i file di output generati nelle elaborazioni dei vari task;
- cancella i file temporanei generati nelle elaborazioni dei vari task, e i file prodotti dai task la cui elaborazione è fallita;
- sposta i file di output dove specificato dall'utente che ha sottomesso la richiesta computazionale.

Queste operazioni sono descritte di seguito.

Copia di un file da nastro a disco

Questa procedura viene attivata quando un file, non presente su disco, deve essere usato come input di un task: è quindi necessario copiarlo su un disk buffer.

Si verifica qual è la dimensione del file (controllando l'attributo 'Size' della tabella *LocalDataFiles*), per poter scegliere un opportuno disk buffer per la memorizzazione: questa operazione viene gestita dal Disk Manager (vedi par. 7.6.12).

Analizzando la tabella *FilesOnTapes*, si individua su che nastro il file considerato è presente. Si "attiva" poi il Local Data Mover, per effettuare la copia del file dal nastro al disk buffer prima individuato (vedi par. 7.6.14).

Quando la copia è stata completata, si inserisce un nuovo elemento nella tabella *FilesOnBuffers*, per definire il file copiato su disco. Infine il Disk Manager aggiorna opportunamente il Local Buffer Database, per registrare il nuovo file copiato su disco (vedi par. 7.6.12).

Definizione di un nuovo file

Questa procedura del Data Manager viene invocata quando si deve definire nel Data Database un nuovo file di dati, che sarà generato da un task pronto per essere eseguito.

Viene creato un nuovo elemento nella tabella *DataFiles*, per definire il file considerato: all'attributo 'FileId' viene associato un nuovo valore. Viene anche inserito un elemento (con stesso FileId) nella tabella *LocalDataFiles*. Il Disk Manager sceglie poi un disk buffer per la memorizzazione del file (vedi par. 7.6.12), e di conseguenza si definisce un corrispondente elemento nella tabella *FilesOnBuffers*.

Registrazione di un file

Quando termina l'elaborazione di una elementary request, è necessario registrare i corrispondenti stream output file (questi file erano già stati definiti nel Data Database prima della loro effettiva creazione). Analogo discorso per i singleton output file, che devono essere registrati quando termina l'esecuzione della relativa richiesta computazionale.

Quando si deve registrare un file, per prima cosa se ne rileva la dimensione reale, in modo da definire l'attributo 'Size' del corrispondente elemento della tabella *LocalDataFiles*. Si invoca poi il Disk Manager, che aggiorna in maniera opportuna il Local Buffer Database (vedi par. 7.6.12).

Cancellazione di un file

I file temporanei di tipo stream e di tipo singleton prodotti nelle varie elaborazioni devono essere cancellati. L'operazione di cancellazione è necessaria anche quando un task non termina con successo la sua esecuzione: prima di rieseguire il task, gli stream file prodotti fino a quel momento devono essere cancellati.

Dopo aver fisicamente cancellato il file da disco, si aggiornano le tabelle *DataFiles*, *LocalDataFiles* e *FilesOnBuffers*, eliminando gli elementi relativi al file considerato. Il Disk Manager aggiorna poi il Local Buffer Database, per "rilasciare" lo spazio prima occupato dal file.

Copia di un file di output

Gli stream output file prodotti in una elementary request, e i singleton output file prodotti in una computational request, devono essere spostati dove indicato dall'utente al momento del submit della richiesta.

Analizzando la tabella *DescriptionsOfOutputsOfRequests*, si verifica dove il file di output considerato deve essere spostato.

Se il file deve essere memorizzato in un disk buffer non si deve fare nulla: si rimuove semplicemente il "lock" associato al file.

Se il file deve essere memorizzato in una directory privata, si esegue la copia su questa directory, mantenendo comunque l'istanza del file presente sul disk buffer (su questo file viene rimosso lo stato di "lock").

Il caso più complesso da considerare è quello per cui il file deve essere memorizzato su un nastro: questa operazione viene gestita dal Tape Manager (vedi par. 7.6.13).

7.6.12 Disk Manager

Il Disk Manager è il modulo che si occupa delle procedure che riguardano i file di dati memorizzati sui disk buffer.
In particolare gestisce le seguenti operazioni:

- scelta di un disk buffer per la memorizzazione di un file;
- modifica del Local Buffer Database per la registrazione di un file;
- modifica del Local Buffer Database per la cancellazione di un file.

Scelta di un disk buffer per la memorizzazione di un file

Il diagramma di figura 7.9 descrive le operazioni che devono essere eseguite dal Disk Manager quando è necessario scegliere un buffer dove memorizzare un file, di cui si conosce la dimensione (o una stima della dimensione).

Si considera l'elemento della tabella *LocalBuffers* che sia "abilitato", e per cui sia più alto il valore del parametro 'FreeSpac' (cioè il disk buffer che ha più spazio libero). Se il valore 'FreeSpac' è comunque inferiore alla dimensione del file che deve essere memorizzato, si considera il file più grande memorizzato su questo buffer, che non sia "marcato" come 'locked': il Tape Manager individua un nastro dove memorizzare questo file (vedi par. 7.6.13), e il Local Data Mover lo sposta da disco a questo nastro (vedi par. 7.6.14). Dopo questa operazione si aggiornano opportunamente le tabelle *LocalBuffers*, *FilesOnBuffers* e *FilesOnTapes*.

Se necessario si ripete la procedura più volte.

Se al termine di questo ciclo il valore del parametro 'FreeSpac' è comunque inferiore alla dimensione richiesta per il file, si ripete l'operazione considerando un altro disk buffer.

Una volta trovato il buffer dove memorizzare il file, si modifica il valore del parametro 'FreeSpac' dell'elemento della tabella *LocalBuffers* relativo a questo disco, sottraendo la dimensione del file (per indicare che su questo buffer si "alloca" uno spazio pari a questo valore).

Se, in ogni caso, l'operazione non ha successo (ovvero non viene trovato un disk buffer che possa essere usato per la memorizzazione del file), il local administrator viene avvertito riguardo il problema. Il Disk Manager continua comunque a controllare se su un disco si rende disponibile lo spazio necessario.

Registrazione di un file di dati

Quando un file di dati deve essere "registrato", sull'elemento della tabella *LocalBuffers* corrispondente al disco dove il file è memorizzato, si modificano i parametri 'FreeSpac' e 'UsSpac', per definire la dimensione reale del file (prima lo spazio era stato solamente allocato).

Cancellazione di un file di dati

Quando un file di dati viene cancellato, sull'elemento della tabella *LocalBuffers* corrispondente al disco dove il file era memorizzato, si modifica il parametro 'FreeSpac', per indicare che lo spazio precedentemente allocato viene rilasciato.

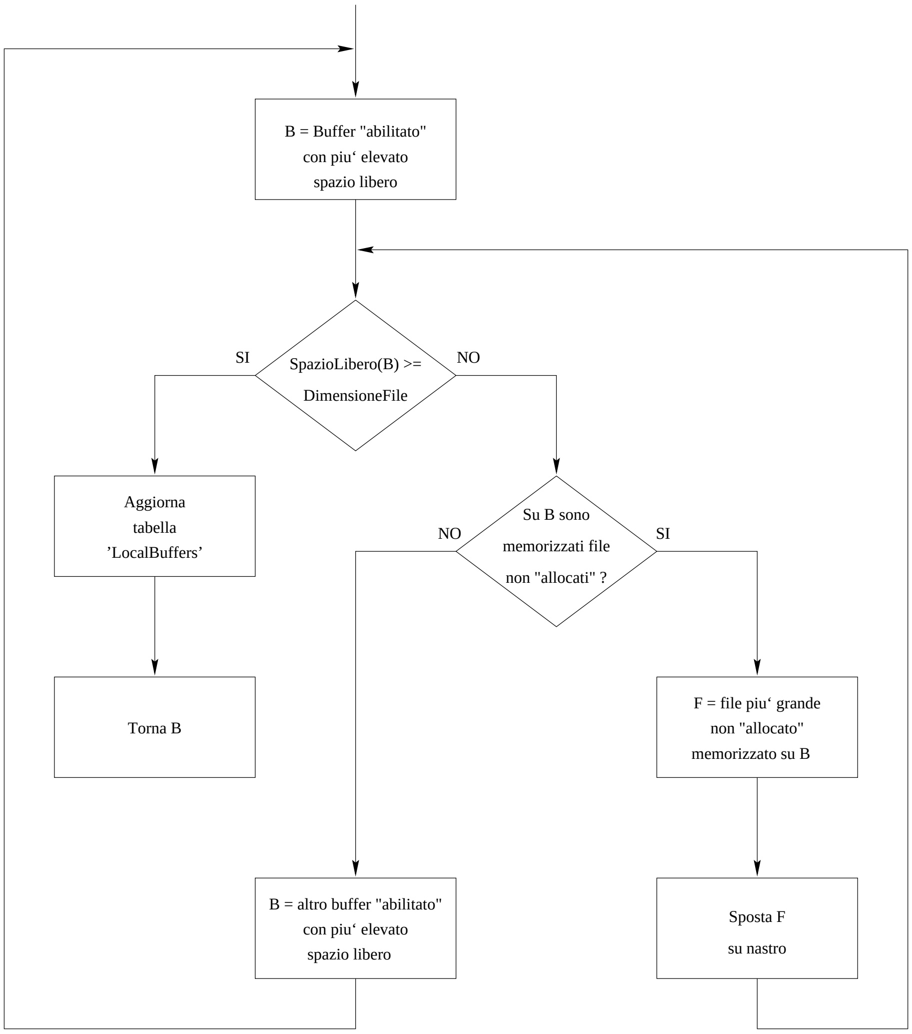


Figura 7.9: Scelta di un disk buffer per la memorizzazione di un file

ReqId	DataBoxId	TapeId
req00001	prod1.out7	tapeid221
req00001	prod1.out8	tapeid342
...	...	...

Tabella 7.3: Tabella 'OutputsOnTapes'

7.6.13 Tape Manager

Il Tape Manager è il modulo che gestisce le operazioni che riguardano i file di dati memorizzati sui nastri.
In particolare questo modulo:

- si occupa di spostare i file di output da disco a nastro;
- individua un nastro dove copiare un file di dati, che deve essere spostato da un disk buffer su cui c'è bisogno di liberare spazio.

Spostamento di un file di output da disco a nastro

È opportuno che, per ogni richiesta computazionale, i corrispondenti file di output vengano memorizzati su "nuovi" nastri (che non contengono cioè altri file). È opportuno inoltre che i file di output associati agli stessi data box siano memorizzati sugli stessi nastri.

Il Tape Manager gestisce allora una struttura dati, chiamata *OutputsOnTapes*, rappresentata in tabella 7.3, in cui ogni elemento è definito da una terna: l'identificatore della richiesta, l'identificatore del data box della corrispondente produzione, e l'identificatore del nastro.

La procedura che il Tape Manager deve considerare quando si deve copiare un file di output da disco a nastro è descritta dal flow chart di figura 7.10.

Si verifica, quindi, se esiste già nella tabella un elemento i cui campi 'ReqId' e 'DataBoxId' corrispondono alla richiesta e al data box associati al file considerato. In caso positivo il file deve essere copiato nel nastro il cui identificatore è rappresentato dal campo 'TapeId'. Si controlla, esaminando l'attributo 'FreeSpac' del corrispondente elemento della tabella *LocalTapes*, se su questo nastro c'è spazio a sufficienza per la memorizzazione del file. In caso positivo si invoca il Local Data Mover, che procede con la copia (vedi par. 7.6.14). Dopo questa operazione viene creato un nuovo elemento nella tabella *FilesOnTapes*, per definire il file copiato su nastro, e si aggiorna il campo 'FreeSpac' del corrispondente elemento della tabella *LocalTapes*.

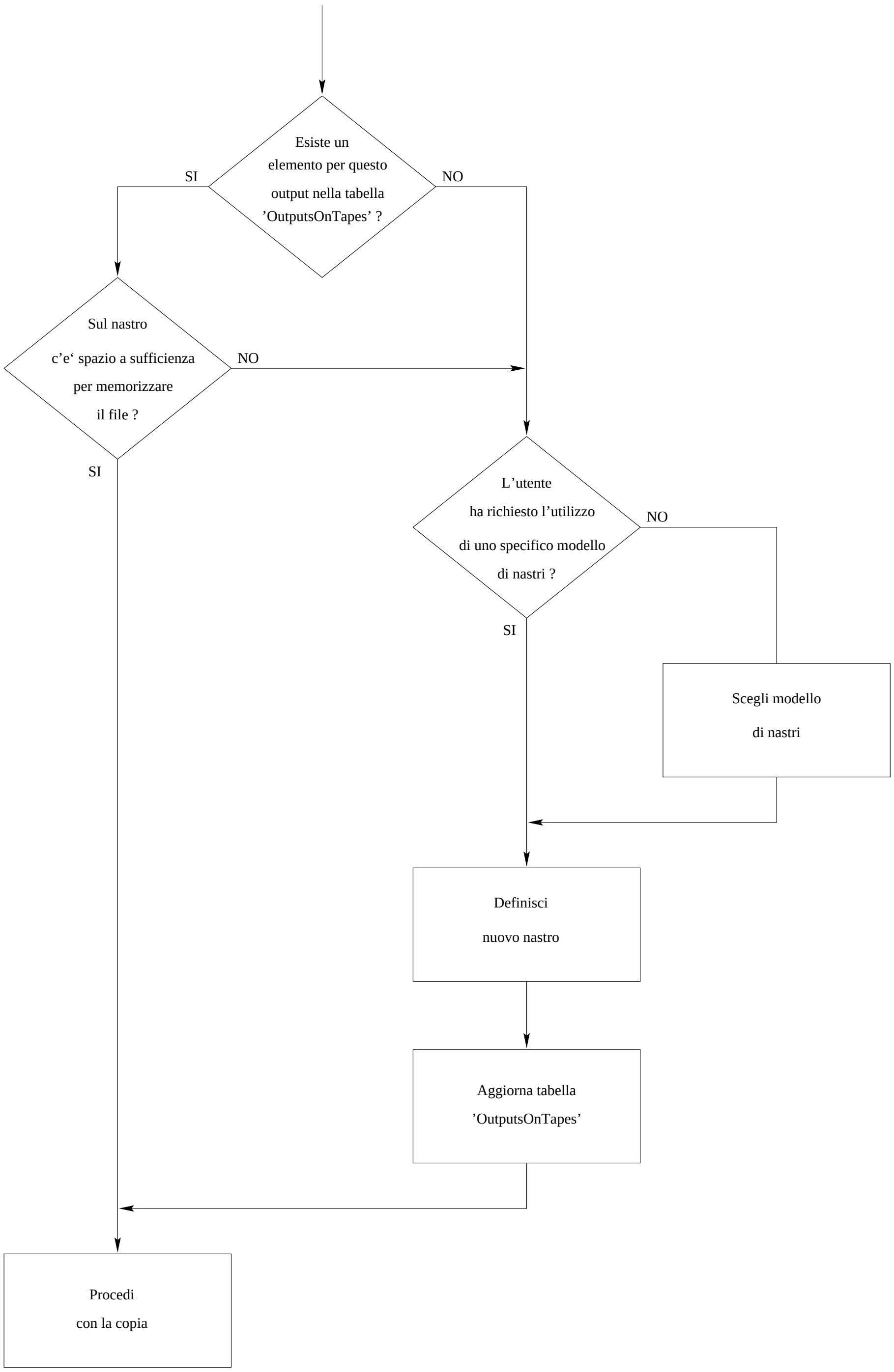


Figura 7.10: Copia di un file di output da disco a nastro

Se invece sul nastro non c'è spazio a sufficienza, se ne deve utilizzare uno nuovo. Prima, però, è necessario scegliere il tipo di nastro da considerare, a meno che l'utente non abbia richiesto l'utilizzo di uno specifico modello (le procedure di scelta del modello di nastro, e di "definizione" di un nuovo nastro sono di seguito descritte). Si può poi aggiornare il campo 'TapeId' della tabella *OutputsOnTapes*, relativamente all'elemento considerato, con l'identificatore del nuovo nastro. Il Local Data Mover procede poi con la copia.

Se invece non esiste un elemento nella tabella *OutputsOnTapes* per questo output (quindi questo è il primo file prodotto associato a questo data box di output, relativamente alla richiesta considerata), lo si deve definire. Anche in questo caso si devono considerare due casi: l'utente può aver definito o meno il tipo di nastro da utilizzare. Nel secondo caso come prima operazione è quindi necessario selezionare un modello. Si definisce poi un nastro del modello considerato.

A questo punto può quindi essere definita la terna (ReqId, DataBoxId, TapeId), che viene inserita nella tabella *OutputsOnTapes*. Il Local Data Mover procede poi con la copia.

Scelta di un modello di nastro

Quando si deve scegliere un modello di nastri (per poi utilizzare un nastro di questo tipo per la memorizzazione di file), si controlla nella tabella *LocalTapeDevices* quali sono i tape device che non sono "disabilitati" (quindi "liberi", o in quel momento usati per una lettura/scrittura da/su nastro): si sceglie il modello di tape device per cui è maggiore il numero di istanze.

Utilizzo di un nuovo nastro

Questa procedura viene invocata quando c'è la necessità di utilizzare un nuovo nastro, di un certo modello specificato, e di cui si indica anche il tipo dei file che dovranno essere memorizzati su di esso (come detto, infatti, i file memorizzati su un nastro devono essere tutti dello stesso tipo).

Viene creato un nuovo elemento della tabella *LocalTapes*, a cui si inizializzano gli attributi 'TapeDevModId' e 'Description'. Al campo 'TapeId' viene semplicemente associato un nuovo valore. Infine l'attributo 'FreeSpac' viene posto pari al valore del campo 'Capacity' del corrispondente elemento della tabella *TapeDeviceModels*.

Il valore dell'identificatore 'TapeId' viene fisicamente scritto su nastro, in modo che lo Staging Manager possa controllare che effettivamente vengano "montati" i nastri richiesti (vedi par. 7.6.14).

Individuazione di un nastro dove copiare un file

Quando si deve individuare un nastro per copiare un file di dati (che deve essere spostato da un disk buffer sul quale è necessario liberare spazio), tra gli elementi della tabella *LocalTapes* se ne seleziona uno, per cui il campo 'Description' coincida con il tipo del file considerato, e per cui ci sia sufficiente spazio per la memorizzazione. Se questo elemento non viene trovato, è necessario "definire" un nuovo nastro. Prima però si deve decidere che modello di nastro considerare (entrambe queste procedure sono state precedentemente descritte).

Il Local Data Mover può poi procedere con la copia.

Il seguente pseudo-codice riassume quanto detto:

```
MoveFromBufferToTape(b : Buffer; f : File);
begin
  typ = DescriptionOfFile(f);
  siz = SizeOfFile(f);
  if (non esiste t nastro con siz spazio libero che contiene file di tipo typ)
    then
          tm = ChooseTapeModel();
          t = NewTape(tm); fi
  CopyToTape(b, f, t);
end
```

7.6.14 Local Data Mover

Il Local Data Mover gestisce la copia di file di dati da dischi a nastri, e viceversa.

Quando si deve copiare un file da uno specifico nastro ad uno specifico disk buffer (o viceversa), per prima cosa si fa richiesta di staging del nastro considerato, ovvero si richiede che tale nastro venga "montato" su un tape device.

Un particolare modulo del sistema DEMI, chiamato **Staging Manager** (rappresentato in figura 7.11), viene utilizzato per la gestione delle richieste di staging.

Lo Staging Manager riceve la richiesta di "mount" di un nastro, e "risponde" a questa richiesta con l'identificatore del tape device sul quale il nastro è stato "montato".

Quando è richiesto di "montare" un certo nastro su un tape device, lo Staging Manager per prima cosa ne identifica il modello, esaminando l'attributo 'TapeDevModId' del corrispondente elemento della tabella *LocalTapes*. Cerca poi nella tabella *LocalTapeDevices* un elemento che rappresenti un tape device di questo modello che sia abilitato, e che in quel momento non sia utilizzato. Se questo

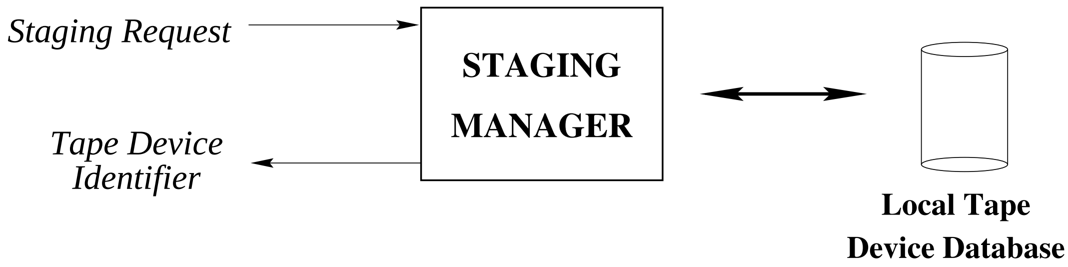


Figura 7.11: Staging Manager

tape device non viene trovato, lo Staging Manager attende finché non se ne rende disponibile uno.

Quando il tape device viene individuato, questo viene "marcato" come allocato, in modo che non venga usato per qualche altra copia. Si cerca poi nella tabella *Users* tutti gli utenti che siano operatori: viene inviato un mail a queste persone, con la richiesta di eseguire il mount del nastro nello specifico tape device considerato.

Non appena il nastro è stato "montato" sul tape device considerato, si effettua la copia del file (da disco a nastro, o viceversa).

Dopo questa operazione, il tape device utilizzato viene definito come "libero" (non più allocato), e quindi disponibile per altre copie.

Lo Staging Manager viene implementato come un processo daemon, che viene "attivato" quando vi sono richieste di "mount" di nastri.

Capitolo 8

Implementazione e verifica

Alla fase di progettazione è seguita la realizzazione di un primo prototipo del sistema DEMI. Obiettivo fondamentale è stato quello di verificare il corretto funzionamento del sistema, e individuare eventuali problemi o anomalie. Comunque sono già stati implementati la maggior parte dei requisiti definiti inizialmente.

In questo capitolo vengono anche presentati alcuni esempi di utilizzo del sistema. I risultati di queste verifiche sono senz'altro positivi: è stata confermata la "validità" del progetto, e sono state individuate le direzioni verso cui proseguire lo sviluppo.

8.1 Implementazione

Il primo prototipo del sistema DEMI è stato implementato su piattaforma Digital Unix, utilizzando il linguaggio C++. Quindi i moduli Global Interface, Local Interface, DEMI Global Control System e DEMI Local Control System possono essere eseguiti su un computer Alpha con sistema operativo Digital Unix. Il porting ad una qualsiasi altra piattaforma Unix, comunque, non comporta nessun problema, in quanto non si sono utilizzate specifiche funzionalità implementate solamente nel sistema operativo Digital Unix. Per l'esecuzione dei vari job, invece, possono essere usati computer di tutte le piattaforme su cui è installato il sistema di load sharing considerato.

Per l'implementazione della base di dati si è impiegato il sistema MySQL [5], un database server relazione di pubblico dominio. L'accesso al database da parte del DEMI Global Control System e del DEMI Local Control System è stato realizzato attraverso l'API ODBC (Open Data Base Connectivity), considerando il driver manager iODBC (intrinsic Open Data Base Connectivity) [2] e il driver ODBC per il sistema MySQL (la descrizione delll'API ODBC è riportata in appendice D).

Come sistema di load sharing si è utilizzato il sistema LSF (presentato nel paragrafo 5.1.2), di cui si sono usate le relative API.

Obiettivo fondamentale nella realizzazione del prototipo è stato quello di verificare la correttezza delle fasi di analisi e progettazione, individuare eventuali problemi e, più in generale, valutare l'effettiva "validità" del sistema complessivo. Comunque già in questo prototipo sono stati implementati la maggior parte dei requisiti definiti inizialmente. Il sistema è infatti in grado di gestire in maniera "automatica" elaborazioni (anche complesse) in un ambiente eterogeneo costituito da workstation Unix e/o PC con sistema operativo Windows NT. L'utente non deve occuparsi direttamente dei vari dettagli implementativi: deve solo definire l'elaborazione, e attendere i risultati da parte del sistema.

Anche il requisito di un sistema self managing è stato realizzato: tutte le informazioni, infatti, vengono gestite in maniera persistente, e quindi, in caso di crash, quando il sistema si "riavvia", esso è in grado di individuare qual era lo stato al momento in cui si è verificato il problema, e può quindi agire di conseguenza.

Non è invece stata ancora realizzata un'interfaccia grafica: al momento l'utente interagisce con il sistema attraverso un'interfaccia di tipo testuale.

8.2 Analisi di un caso reale

Per verificare l'effettiva funzionalità del sistema, lo si è voluto innanzittutto provare con un'applicazione reale di fisica delle alte energie.

Viene di seguito descritto il tipo di elaborazione considerata, analizzato l'ambiente di esecuzione, per presentare infine i risultati.

8.2.1 Descrizione dell'elaborazione

In questo primo test si è considerata una classica elaborazione simulazione-ricostruzione dell'esperimento DELPHI.

La produzione utilizzata è mostrata in figura 8.1.

Il task *Delsim* è il task che gestisce la fase di simulazione. All'input *Inp1*, di tipo singleton, viene associato un file di titoli, che definisce il tipo, il numero di eventi da generare, e altri parametri. Il data box *Out1*, di tipo stream, rappresenta i raw data simulati generati da *Delsim*. All'inizio viene generato un numero casuale, che viene usato nel corso dell'elaborazione di Delsim: in questo modo viene generata una sequenza di file "diversi". I dati generati da Delsim vengono usati dal task *Delana*, che effettua la ricostruzione, producendo così eventi simulati ricostruiti,

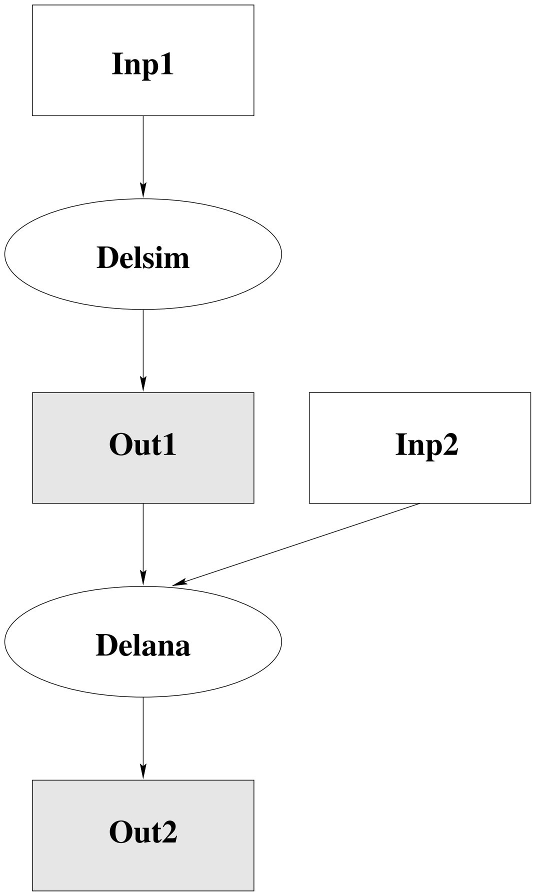


Figura 8.1: La produzione considerata nel primo test

associati al data box *Out2*. *Delana* ha anche un secondo input (rappresentato dal data box *Inp2*): ad esso viene associato un file di titoli che definisce alcune costanti che devono essere considerate nell'elaborazione.

In questo test si sono considerate due architetture: Digital-Unix e Hp-Ux: i task *Delsim* e *Delana* sono stati implementati per entrambe le piattaforme. Il "costo" dei task è lo stesso per le due architetture, poiché il tempo di esecuzione dipende esclusivamente dalle prestazioni dei computer, e non da caratteristiche specifiche del tipo di architettura.

Considerando questa produzione, si è sottomessa una richiesta computazionale. Non essendovi input di tipo stream si è specificato il numero di iterazioni: è stato scelto un valore pari a 80. All'input di tipo singleton *Inp1* si è associato come singleton input file un file di titoli, in cui si indica di generare 50 eventi di tipo QQPS (quindi in totale nelle 80 iterazioni vengono prodotti e ricostruiti 4000 eventi).

Mediamente per eseguire *Delsim* su un'alpha station 3000/600, per la produzione di 50 eventi QQPS, sono necessari circa 100 minuti, mentre per la ricostruzione (attraverso *Delana*) di questi 50 eventi su un calcolatore dello stesso modello, sono necessari circa 30 minuti. Quindi, se si considerasse un'elaborazione sequenziale su una singola alpha station 3000/600 utilizzata a tempo pieno, sarebbero necessari più di sette giorni.

Per quel che riguarda la dimensione massima dei singoli file associati ai data box di output, questa è pari a 28 Mbyte per il data box *Out1*, e 6 Mbyte per il data box *Out2*.

8.2.2 Descrizione dell'ambiente

Per l'esecuzione si è considerato l'ambiente rappresentato in figura 8.2.

Si sono utilizzate in totale otto workstation, di due diverse architetture: sei alpha station con sistema operativo Digital Unix 4.0B (architettura *AXP-DU*), e due workstation HP con sistema operativo HP-UX 10.20 (architettura *HP-UX*). Il Local Scheduler è stato installato su un'altra alpha station. In tabella 8.1 sono riportate le caratteristiche di questi computer (il campo 'Performances' rappresenta il corrispondente attributo del Local Computer Database, e i relativi valori fanno riferimento ai CERN benchmark).

Per la memorizzazione dei file si sono utilizzati due disk buffer, di capacità rispettivamente di 2 Gbyte e 1 Gbyte, fisicamente installati su due alpha station. Questi disk buffer sono stati condivisi con le altre workstation (sia Digital che HP) via NFS.

Per quel che riguarda la configurazione del batch system LSF, si è definita

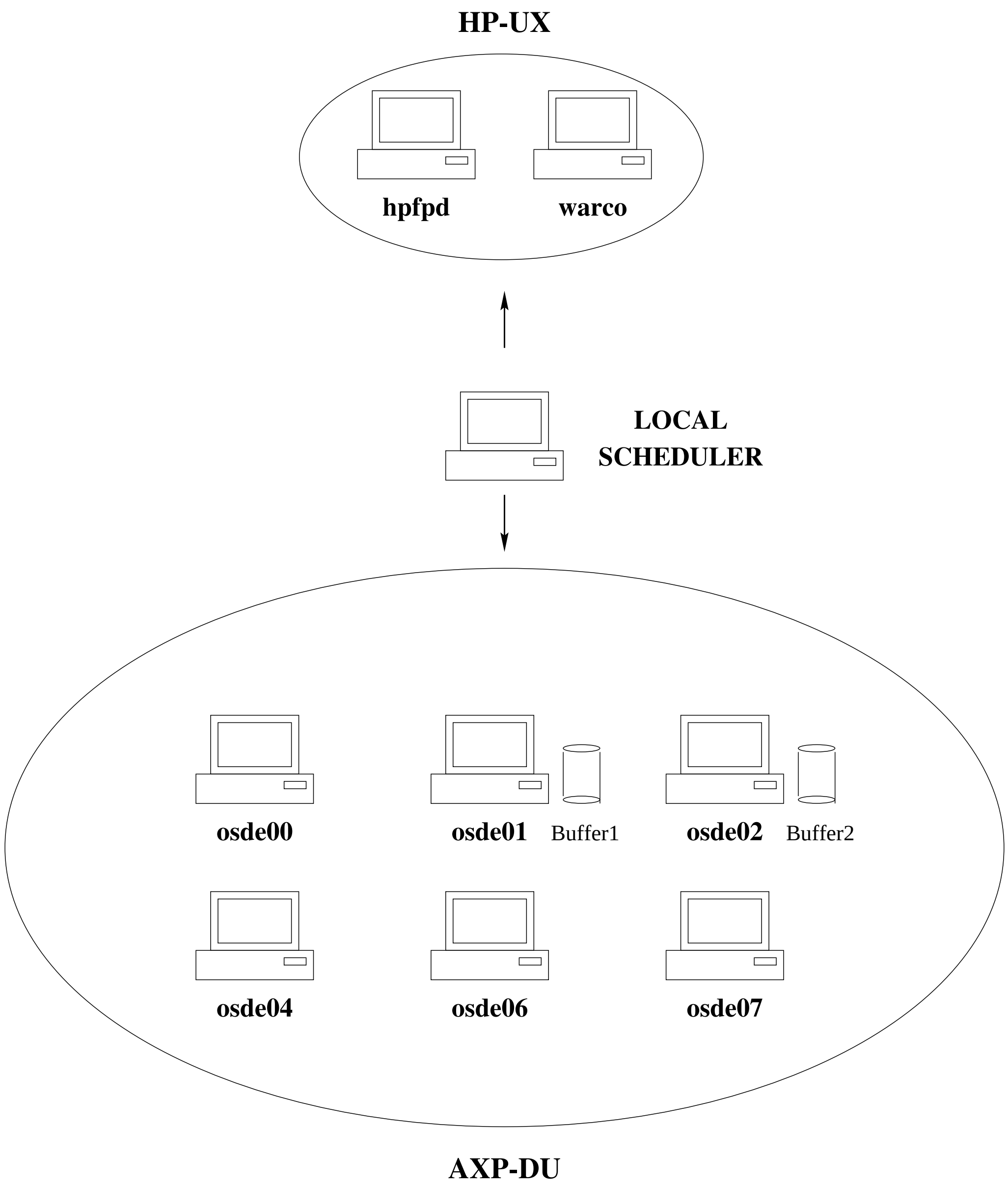


Figura 8.2: Ambiente considerato nel primo test

Nome	Modello	Sistema Operativo	Architettura	Performances
osde00	Alpha Station 600 5/266 128 MB	Digital Unix 4.0B	AXP-DU	34
osde01	Alpha Station 3000/600 64 MB	Digital Unix 4.0B	AXP-DU	12
osde02	Alpha Station 3000/200 64 MB	Digital Unix 4.0B	AXP-DU	7
osde04	Alpha Station 3000/400 64 MB	Digital Unix 4.0B	AXP-DU	9
osde06	Alpha Station 3000/600 64 MB	Digital Unix 4.0B	AXP-DU	12
osde07	Alpha Station 3000/600 64 MB	Digital Unix 4.0B	AXP-DU	12
hpfpd	HP Workstation 9000/735 96 MB	HP-UX 10.20	HP-UX	12
warco	HP Workstation 9000/735 96 MB	HP-UX 10.20	HP-UX	12

Tabella 8.1: Computer usati nel test

un'unica coda per tutte queste workstation, configurata in modo che i job del sistema DEMI possono essere eseguiti su un computer solamente se esso non è usato per altre elaborazioni batch, o in maniera interattiva. Più un dettaglio ai job di questa coda è stata assegnata una priorità più bassa rispetto a quelli delle altre code, e questi job vengono eseguiti con un valore di nice pari a 20. Non ci possono essere due o più task DEMI in esecuzione contemporaneamente. Un job di questa coda può iniziare l'esecuzione (o se ne può fare il resume da uno stato di suspend) se la media, calcolata sull'ultimo minuto, del numero di processi in attesa di usare il processore è minore o uguale a 1, e se non c'è attività interattiva da almeno cinque minuti. Un job viene invece sospeso quando la media dell'ultimo minuto del numero di processi in attesa di utilizzare il processore è maggiore o uguale a 2.5, o quando c'è attività interattiva da almeno un minuto.

Non si è utilizzato il meccanismo del checkpointing: quindi viene gestito il suspend e il resume dei job, che però non vengono "migrati" tra i diversi computer.

8.2.3 Analisi dei risultati

L'elaborazione è iniziata alle ore 12:15 di un giovedì. L'ultimo task ha completato la sua esecuzione alle ore 17.15 del giorno successivo (quindi 29 ore dopo).

Nelle figure 8.3 e 8.4 si riporta su che calcolatori sono stati eseguiti rispettivamente gli 80 task *Delsim* e gli 80 task *Delana*.

Nelle figure 8.5 e 8.6 si indica invece quando questi task sono stati completati sui vari computer.

La tabella 8.2 indica, per i vari calcolatori, il tempo totale speso per l'esecuzione dei task *Delsim/Delana* (il campo 'Tempo esecuzione'), e il tempo invece in cui i task sono rimasti in stato di suspend (il campo 'Tempo susped') poiché i relativi computer erano usati per altre attività (sostanzialmente uso interattivo da parte degli utenti). Il grafico di figura 8.7 mostra quando questi computer hanno elaborato i task *Delsim/Delana* (linee verdi), e quando questi task sono rimasti in stato di suspend (linee rosse).

Come ci si poteva aspettare, in genere i job sono stati in esecuzione ininterrottamente di notte, mentre di giorno le workstation sono spesso state usate dagli utenti (per *osde00*, la workstation a più elevate prestazioni, lo stato di suspend è stato prevalente rispetto allo stato di esecuzione).

Le prestazioni misurate sono piuttosto elevate. Per la simulazione e ricostruzione il throughput è stato infatti di circa 3300 eventi al giorno mentre, se si fosse utilizzata a tempo pieno una singola alpha station 3000/600, il throughput sarebbe stato di circa 550 eventi al giorno (quindi con il sistema DEMI è stato fatto il lavoro di sei alpha station 3000/600, utilizzate a tempo pieno). È importante sottolineare

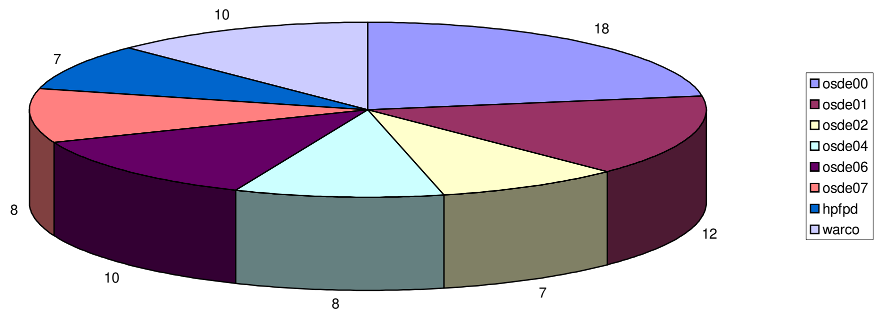


Figura 8.3: Task *Delsim* eseguiti nei vari calcolatori

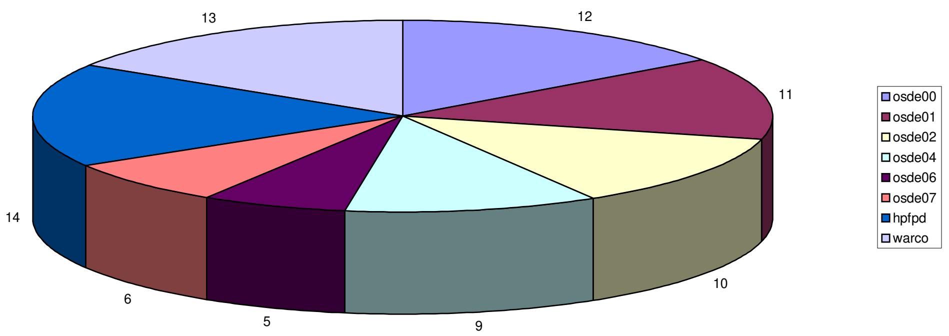


Figura 8.4: Task *Delana* eseguiti nei vari calcolatori

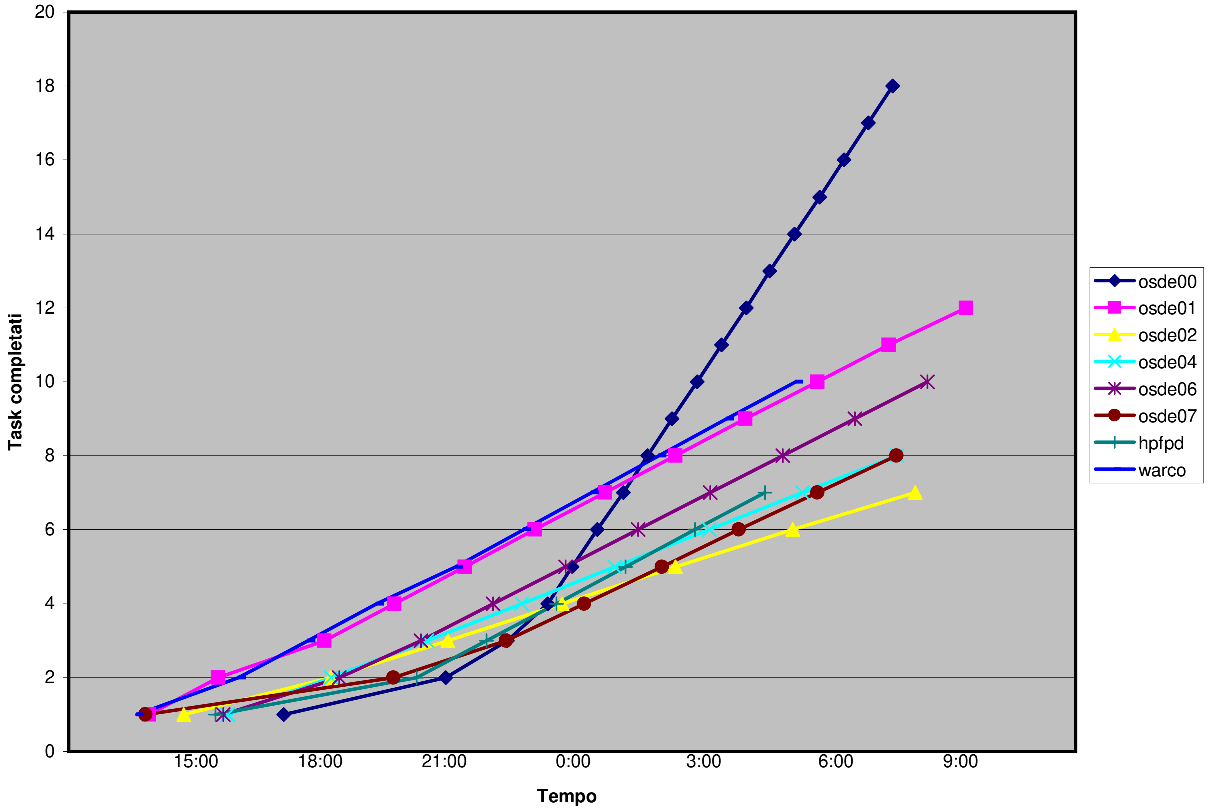


Figura 8.5: Task *Delsim* completati sui vari calcolatori

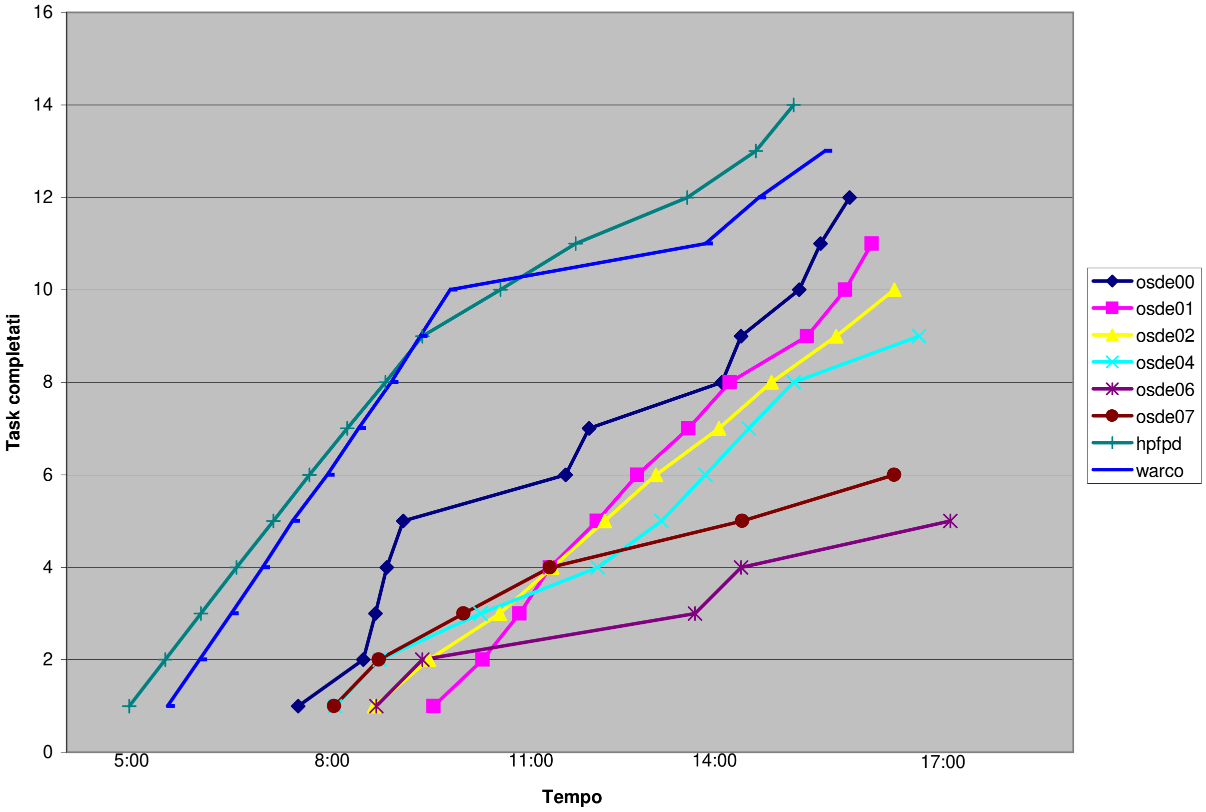


Figura 8.6: Task *Delana* completati sui vari calcolatori

Computer	Tempo esecuzione	Tempo suspend
osde00	47503 s.	51379 s.
osde01	90445 s.	9780 s.
osde02	99987 s.	1060 s.
osde04	84330 s.	18171 s.
osde06	71267 s.	32930 s.
osde07	66425 s.	34640 s.
hpfpd	69735 s.	25762 s.
warco	79576 s.	18175 s.
	609268 s.	191897 s.

Tabella 8.2: Utilizzo dei vari calcolatori nel primo test

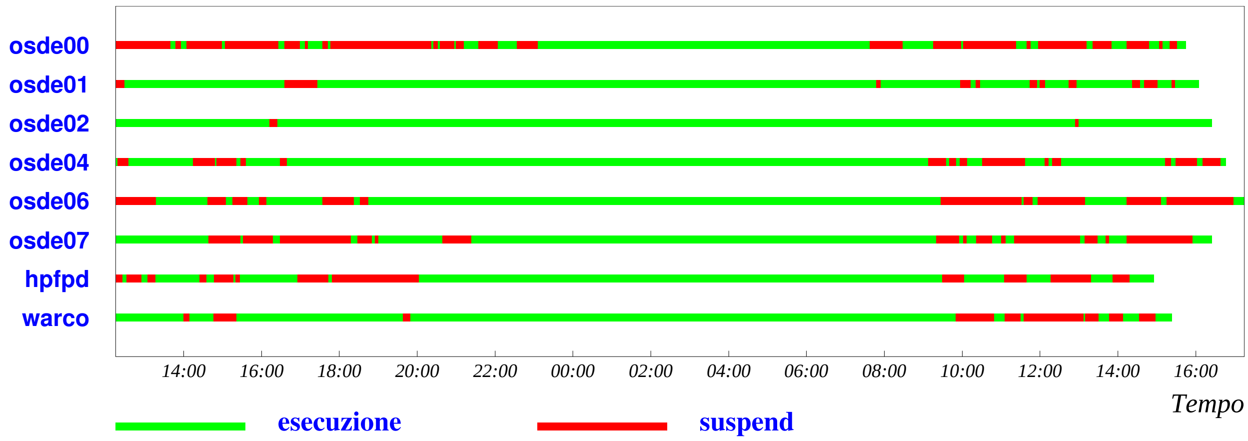


Figura 8.7: Esecuzione e suspend dei job DEMI

che nel test non si è interferito minimamente con le attività degli utenti, che hanno potuto usare senza problemi le varie workstation.

Questi risultati potrebbero essere ulteriormente migliorati con alcuni raffinamenti. Dal grafico si può notare come le elaborazioni sulle workstation dell'architettura Hp-Ux (*hpfpd* e *warco*) siano terminate prima rispetto alle elaborazioni sulle alpha station. Il problema è dovuto a come viene effettuato lo scheduling locale: il Local Scheduler (e in particolare il modulo che decide a che architetture assegnare i task da eseguire) viene "attivato" quando c'è un'elementary request che deve essere elaborata. Decidere a che architettura assegnare un job da eseguire, quando ci sono già centinaia o migliaia di altri job in coda, non è la scelta ottimale, in quanto non si può sapere come evolverà l'elaborazione: non si può prevedere quando e per quanto tempo i vari calcolatori delle diverse architetture saranno disponibili, o usati dagli utenti. Si potrebbe allora modificare questa parte del sistema DEMI, in modo da decidere a che architettura assegnare un task, solo quando c'è un calcolatore "libero", pronto per eseguire elaborazioni[1].

Può succedere, inoltre, che vi siano dei lunghi intervalli di tempo tra il termine delle elaborazioni dei job DEMI sui computer della stessa architettura. Questo problema è legato all'elaborazione dell'ultimo job su ogni computer: il processing può procedere ininterrottamente, oppure il job, una volta iniziata l'elaborazione, può restare in stato di 'suspend' per un lungo periodo. Si potrebbe ovviare a questo inconveniente utilizzando il meccanismo del checkpointing: se un job resta in stato di suspend per un lungo periodo, può essere migrato su un altro calcolatore disponibile. D'altro canto bisogna ricordare che salvare periodicamente su disco lo stato dei vari job può comportare un sensibile degrado delle prestazioni.

8.3 Altro esempio

È stata verificata la funzionalità del sistema considerando anche altre applicazioni.

In questo paragrafo si esamina un secondo test del prototipo, in cui si è definita appositamente una produzione più "articolata" rispetto a quella del test precedente, e in cui si sono utilizzati per l'esecuzione calcolatori di diverse architetture (Digital-Unix, Hp-Ux, Windows-NT). Anche se non si tratta di un'applicazione reale, si ritiene quindi questa prova molto significativa, vista la complessità dell'elaborazione e dell'ambiente di calcolo.

Si sono considerati tre tipi di task, rappresentati in figura 8.8, utilizzati poi per definire la produzione.

[1] Per verificare quando un computer è disponibile, si possono usare le apposite procedure del sistema di load sharing

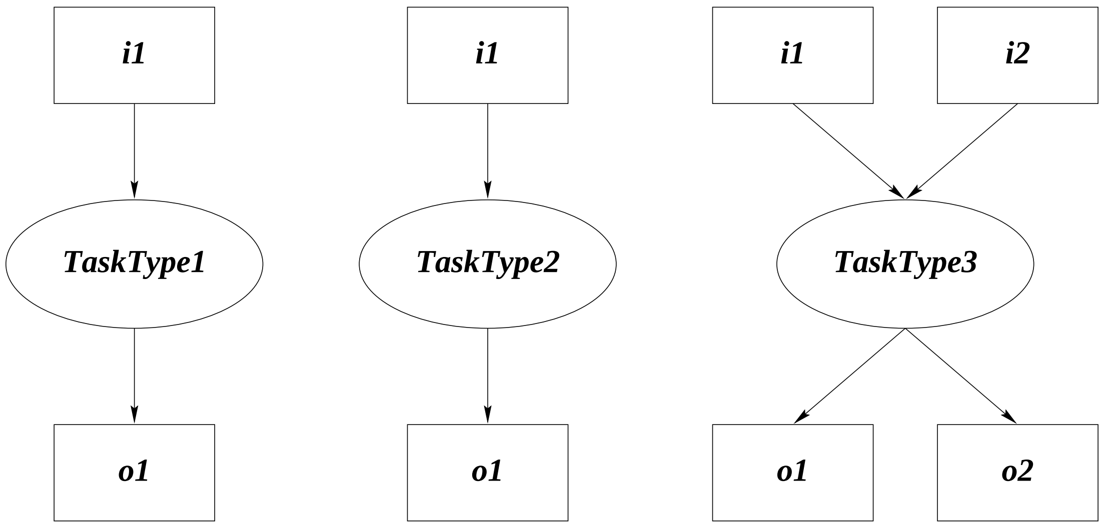


Figura 8.8: Tipi di task utilizzati nel secondo test

Il tipo di task *TaskType1* e il tipo di task *TaskType2* hanno un input e un output, mentre il tipo di task *TaskType3* ha due input e due output.

In questi task si legge il valore degli input, e si eseguono delle computazioni che fanno uso intenso di CPU. Infine vengono creati i corrispondenti file di output. Per quel che riguarda l'output *o1* del tipo di task *TaskType3*, non viene creato un nuovo file, ma viene fatto un append ad un file già esistente (il file viene creato solo la prima volta).

Approssimativamente il tempo necessario per l'esecuzione di un task di tipo *TaskType2* è circa un terzo rispetto al tempo richiesto per un task di tipo *TaskType1*, e metà rispetto ad un task di tipo *TaskType2*. Considerando ad esempio un'alpha station 3000/600, per eseguire un task di tipo *TaskType1*, un task di tipo *TaskType2*, e un task di tipo *TaskType3* sono necessari rispettivamente circa 18 minuti, 6 minuti e 12 minuti.

Per rendere più significativo il test, non si sono implementati tutti i tipi di task per tutte le architetture disponibili: il tipo di task *TaskType1* è stato realizzato per le architetture Hp-Ux e Windows-NT, il tipo di task *TaskType2* è stato implementato per le architetture Digital Unix, Hp-Ux e Windows-NT, mentre il tipo di task *TaskType3* è stato realizzato per le architetture Digital Unix e Hp-Ux.

Usando questi tre tipi di task, si è definita la produzione mostrata in figura 8.9, costituita da quattro task: *Task1* di tipo *TaskType1*, *Task2* di tipo *TaskType2*, *Task3* di tipo *TaskType3*, e *Task4* di tipo *TaskType2*.

Per quel che riguarda i data box, *Inp1* e *Inp3* sono input di tipo singleton. *Out2*, *Out4* e *Out7* sono output di tipo stream, mentre *Out5* è un output di tipo

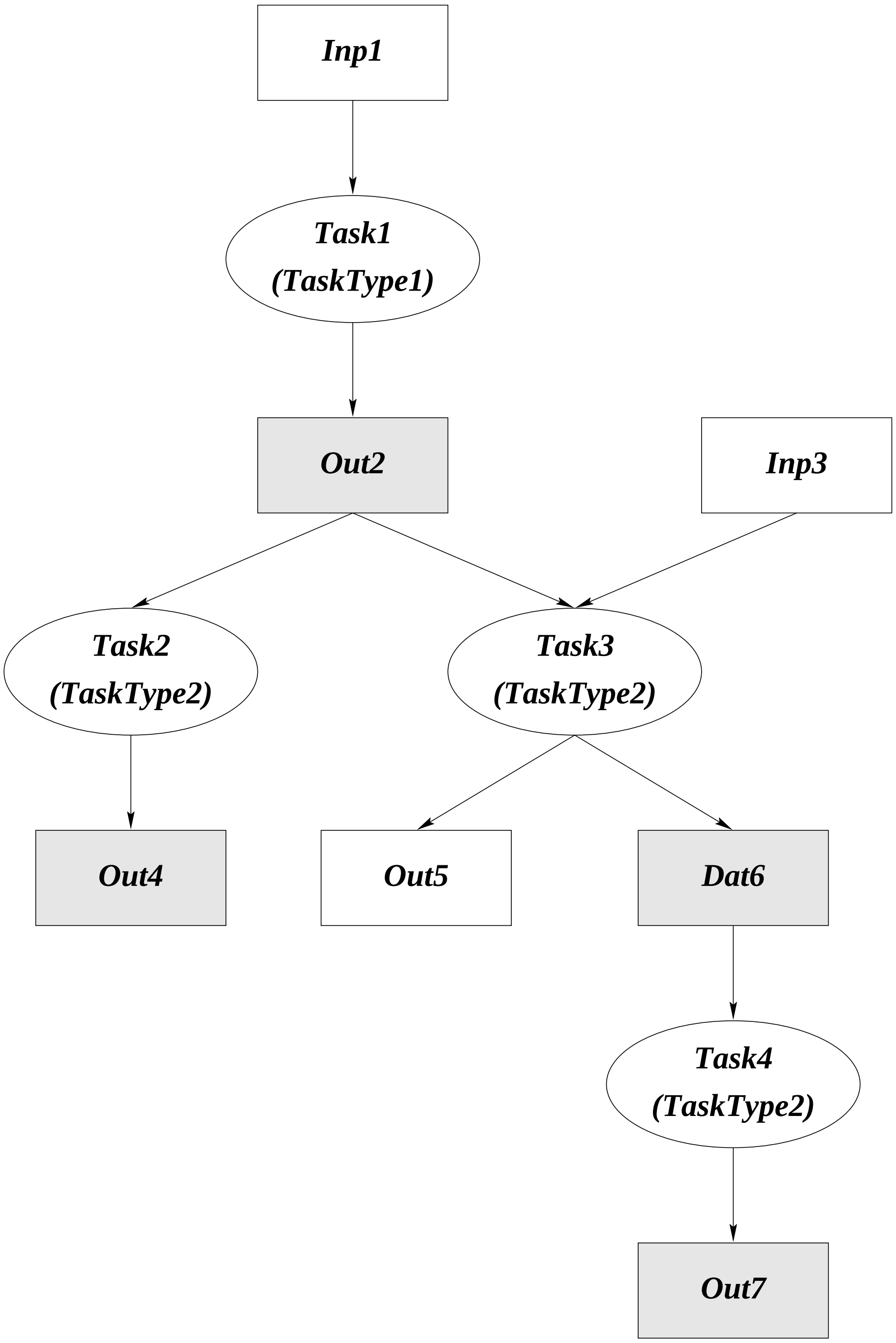


Figura 8.9: Produzione considerata nel secondo test

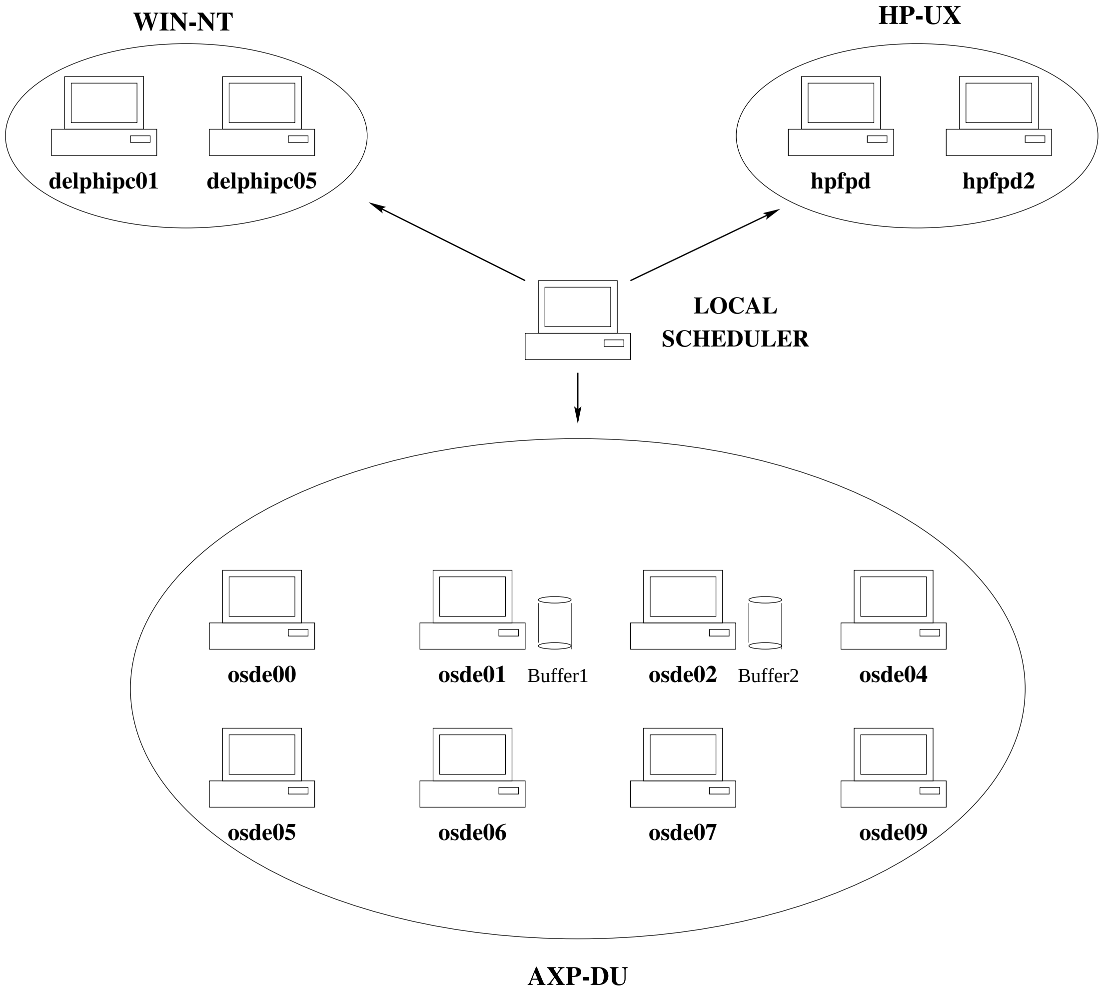


Figura 8.10: Ambiente considerato nel secondo test

singleton. Infine *Dat6*, di tipo stream, rappresenta dati temporanei (i relativi file vengono quindi cancellati dopo essere stati usati come input dal task *Task4*).

Considerando questa produzione, si è "sottomessa' una richiesta computazionale: si è specificato un numero di iterazioni pari a 300 (in totale, quindi, sono stati eseguiti 1200 task: 4 task per ogni iterazione).

Per quel che riguarda la dimensione massima dei singoli file associati ai vari data box della corrispondente produzione, questa è pari a 1 Mbyte per i file associati ai data box *Inp1*, *Inp3*, *Out2* e *Out5*, 2 Mbyte per i file associati ai data box *Out4* e *Out7*, e 4 Mbyte per i file associati al data box *Dat6*.

Per l'esecuzione si è considerato l'ambiente rappresentato in figura 8.10.

Si sono utilizzate in totale dodici workstation, di tre diverse architetture:

- otto alpha station con sistema operativo Digital Unix 4.0B (architettura *AXP-DU*), di cui:
  - un'alpha station 600 5/266: *osde00*;
  - cinque alpha station 3000/600: *osde01*, *osde05*, *osde06*, *osde07*, *osde09*;
  - un'alpha station 3000/400: *osde04*;
  - un'alpha station 3000/300: *osde02*;
- due workstation HP con sistema operativo HP-UX 10.20 (architettura *HP-UX*), di cui:
  - una workstation HP 9000/780: *hpfpd2*;
  - una workstation HP 9000/735: *hpfpd*;
- due PC Pentium Pro 200 con sistema operativo Windows NT 4 Workstation: *delphipc01* e *delphipc05* (architettura *WIN-NT*).

Si sono considerati due disk buffer, di capacità rispettivamente 1.1 Gbyte e 950 Mbyte, fisicamente installati su due alpha station. Questi disk buffer sono stati condivisi con le altre workstation Unix (sia Digital che HP) via NFS, mentre per i PC Windows NT si è utilizzato Samba.

Per quel che riguarda il load sharing system LSF, si è considerata la stessa configurazione del primo test.

L'elaborazione è iniziata alle ore 20:30 di un lunedì sera. L'ultimo task ha completato la sua esecuzione alle ore 12:45 del giorno successivo (quindi circa 16 ore dopo).

Non vengono analizzati in dettaglio i risultati dell'elaborazione (visto che non si tratta di un'applicazione reale, in questo test si era più interessati a valutare la funzionalità del sistema, considerando un'elaborazione e un ambiente di esecuzione complessi): si riporta solo, nelle figure 8.11, 8.12, 8.13, e 8.14, su che calcolatori sono stati eseguiti rispettivamente i task *Task1*, *Task2*, *Task3* e *Task4*.

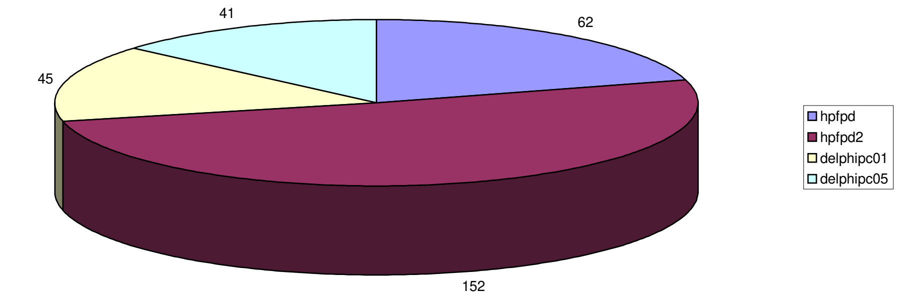


Figura 8.11: Task *Task1* eseguiti nei vari calcolatori

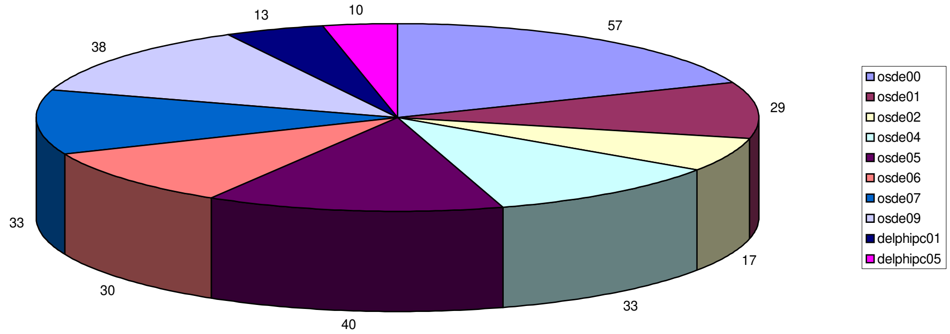


Figura 8.12: Task *Task2* eseguiti nei vari calcolatori

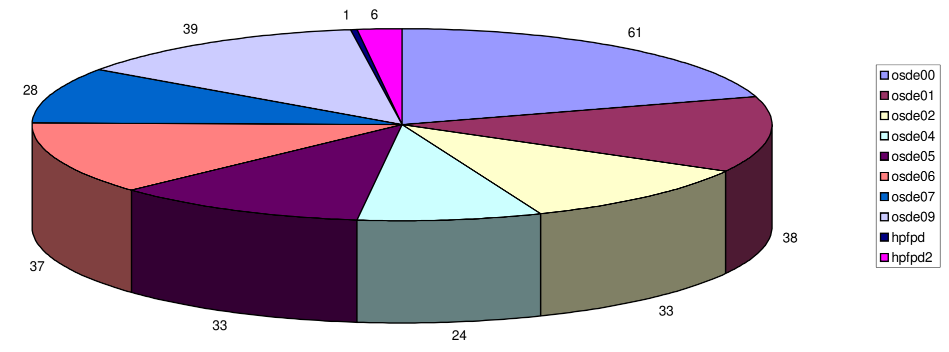


Figura 8.13: Task *Task3* eseguiti nei vari calcolatori

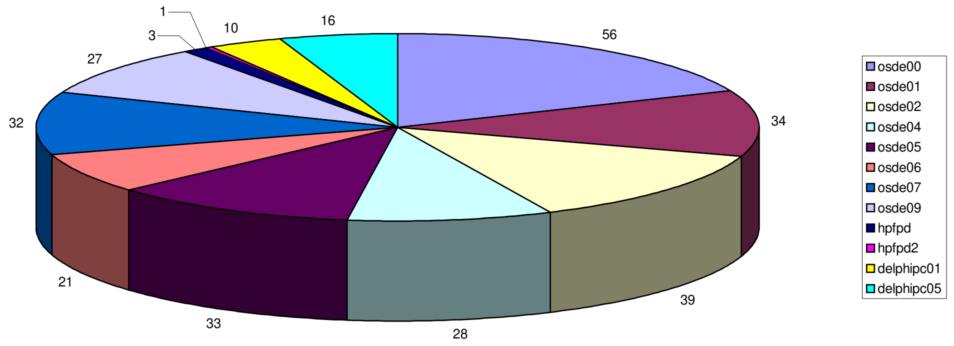


Figura 8.14: Task *Task4* eseguiti sui vari calcolatori

8.4 Conclusioni

È stato quindi verificato il corretto funzionamento del sistema, considerando diversi tipi di elaborazioni, sia reali, che costruite appositamente per esaminare casi più complessi. Negli esempi descritti si sono analizzate "singole" produzioni, ma si è verificata la funzionalità del sistema considerando anche diversi tipi di elaborazioni, sottomesse contemporaneamente al sistema.

Si sono considerati anche diversi ambienti di esecuzione, costituiti da calcolatori di architetture diverse. In particolare si è verificata la piena integrazione tra le workstation Unix, e i PC Windows NT.

Per quel che riguarda l'interoperabilità tra il sistema DEMI e il load sharing system LSF, non si sono riscontrati particolari problemi o anomalie. È probabilmente opportuna una maggiore "integrazione" tra i vari task definiti dagli utenti, e la configurazione del pool di workstation (ad esempio per definire vincoli del tipo: il task *taskx* può essere eseguito solamente sui calcolatori che hanno una memoria libera almeno pari a y, o per decidere se usare o meno il meccanismo del checkpointing, a seconda dei casi). Questi raffinamenti, comunque, comporterebbero piccole modifiche del codice, "isolate" a specifici moduli.

I risultati dei test sul prototipo del sistema DEMI sono quindi da considerarsi senz'altro positivi. Le verifiche sono state utili anche perché si sono potuti individuare gli elementi che, se raffinati, possono comportare un ulteriore miglioramento delle prestazioni complessive del sistema.

Capitolo 9

Conclusioni

In questa tesi è stata presentata l'analisi e la progettazione di un sistema, chiamato DEMI, per la programmazione di job, costituiti da unità computazionali elementari opportunamente combinate tra loro. Il sistema gestisce poi l'esecuzione di questi job in un ambiente di calcolo distribuito ed eterogeneo.

Il sistema è stato progettato soprattutto per poter gestire le elaborazioni tipiche dell'ambiente della fisica delle alte energie (senza riferimento ad uno specifico esperimento), ma esso è comunque facilmente "esportabile" anche ad altri domini.

È stato poi realizzato un primo prototipo del sistema, che già implementa la maggior parte dei requisiti definiti inizialmente. Il sistema è infatti in grado di gestire in maniera efficiente elaborazioni, anche complesse, in un ambiente costituito da workstation Unix e/o PC con sistema operativo Windows NT. Il sistema utilizza i diversi calcolatori in maniera uniforme e del tutto trasparente per gli utenti, che considerano DEMI come un unico, potente sistema di calcolo.

È importante sottolineare come, a differenza di sistemi già esistenti, DEMI è flessibile, scalabile, e facilmente configurabile: il sistema infatti non è specifico ad un particolare ambiente di calcolo, o ad un particolare tipo di elaborazione, e può essere adattato per le specifiche esigenze che si devono considerare.

I vantaggi più evidenti sono comunque per l'utente finale, che deve solo preoccuparsi di definire l'elaborazione da eseguire, e attendere che gli vengano forniti i relativi risultati: non deve preoccuparsi di gestire i vari dettagli implementativi. Altro elemento di novità è che anche l'attività di gestione da parte dell'amministratore è minima: le varie funzionalità sono del tutto "automatizzate". Il sistema è anche self managing, in quanto è in grado di gestire autonomamente eventuali failure, senza l'intervento dell'amministratore.

Con riferimento all'ambiente della fisica delle alte energie, le elaborazioni che si devono considerare richiedono spesso molte risorse (specialmente cpu e memoria), e ci sono quindi problemi ad usare le workstation utilizzate dagli utenti per questi processing. Attualmente ci sono due possibili approcci per l'esecuzione di questi job: acquistare workstation da dedicare solo a queste attività, o usare le workstation degli utenti solo in determinati periodi, quando si prevede che non ci sia attività di tipo interattivo (sostanzialmente di notte e nei fine settimana). In entrambi i casi ci sono problemi da considerare: il costo da sostenere per l'acquisto di nuovi calcolatori, il fatto che comunque di notte e nei weekend ci possono essere elaborazioni degli utenti, l'utilizzo non ottimale dei vari computer, le cui potenzialità non vengono del tutto sfruttate.

Con il sistema DEMI, grazie alla sua integrazione con un sistema di load sharing, questi problemi vengono superati: si garantisce il massimo utilizzo dei vari calcolatori, senza comunque interferire con le normali attività degli utenti.

9.1 Sviluppi futuri

Il sistema realizzato può essere completato e integrato sotto diversi aspetti.

Potranno essere fatte, innanzittutto, modifiche e raffinamenti per migliorare l'affidabilità e l'efficienza complessiva. Ad esempio, come spiegato nel capitolo precedente, può essere modificato l'algoritmo di scheduling locale, in modo da ottimizzare le prestazioni del sistema.

È poi necessario implementare un'adeguata interfaccia grafica, in modo da rendere semplice l'utilizzo del sistema da parte degli utenti e degli amministratori. Particolarmente curata dovrà essere l'implementazione dell'interfaccia grafica per la definizione di task e di produzioni. Potrà essere usato il formalismo proposto in questa tesi, rappresentando quindi i task attraverso ellissi, i data set attraverso rettangoli, e le produzioni attraverso grafi, in cui i vari task vengono combinati tra loro.

Altro possibile lavoro futuro è il porting dei vari moduli del sistema (Global Interface, Local Interface, DEMI Global Control System, DEMI Local Control System) alla piattaforma Windows NT: questa non dovrebbe essere, comunque, un'attività molto complessa, in quanto non si sono utilizzati particolari feature implementate solamente per le architetture Unix.

Si può poi prevedere l'integrazione del sistema DEMI con altri sistemi di load sharing, oltre a LSF, che abbiano comunque tutte le funzionalità richieste per la gestione di batch job.

È poi necessario l'impiego di un DBMS più efficiente e affidabile rispetto al sistema MySQL utilizzato nel prototipo realizzato. Questo DBMS deve anche essere in grado di gestire grossi volumi di informazioni. In praticamente tutti gli esperimenti di fisica delle alte energie si è scelto di adottare, per la memorizzazione dei vari dati, *Objectivity/DB*, un DBMS a oggetti, implementato su molte piattaforme (la maggior parte dei sistemi Unix, e Windows NT), e che permette di definire database distribuiti tra più calcolatori, posti anche in siti diversi [42]. Attualmente è in fase di definizione un'interfaccia tra Objectivity/DB e *HPSS* (*High Performance Storage System*) [58], un sistema software in grado di gestire in maniera gerarchica un volume di dati molto elevato (dell'ordine dei Pbyte), con ottime prestazioni [56]. Si può quindi valutare se Objectivity/DB può essere un utile strumento da impiegare nel sistema DEMI.

Per completare la realizzazione del sistema sarà infine necessario progettare e implementare il Global Scheduler, in modo da poter gestire le elaborazioni in un ambiente di calcolo distribuito geograficamente tra più siti.

Saranno poi gli utenti finali a suggerire migliorie o nuove funzionalità da implementare.

Ringraziamenti

Sono molte le persone che voglio ringraziare, in quanto, in forme e modi diversi, hanno dato il loro contributo alla realizzazione di questo lavoro.

Innanzittutto voglio ringraziare il relatore, dott. Salvatore Orlando, e i correlatori, il dott. Mirco Mazzucato e il prof. Roberto Zannoni, sempre disponibili a mettere a disposizione le loro competenze, e che mi hanno dato numerosi consigli e suggerimenti, di cui spero aver fatto buon uso.

Desidero ringraziare anche il dott. Jean Claude Marin e il dott. Franco Carena del Cern di Ginevra: dalle discussioni che ho avuto con loro ho sempre imparato qualcosa di nuovo.

Un caloroso ringraziamento anche a tutti i docenti del corso di laurea in scienze dell'informazione: in questi anni sono sempre stati tutti estremamente disponibili nel dedicarmi parte del loro tempo, andando ben oltre i loro obblighi.

Un ringraziamento speciale per i miei colleghi e amici del servizio calcolo dell'INFN di Padova: Federico Balbi, Roberto Ferrari, Massimo Gravino, Fabio Montecassiano, Ivo Saccarola, Lorenzo Turicchia, e in particolare Fulvia Costa: lavorare con loro è sempre stata ed è tuttora un'esperienza estremamente positiva. Sono grato anche a quelli che sono stati i responsabili del servizio calcolo in questi anni: il prof. Maurizio Morando, il dott. Roberto Stroili, il dott. Mauro Morandin e il dott. Michele Michelotto, che, nonostante i miei impegni di lavoro, mi hanno sempre sostenuto nel proseguimento degli studi.

Ringrazio poi tutti gli altri amici dell'INFN di Padova, in particolare Gabriele Rampazzo e Mario Zago che, oltre ad offrire la loro simpatia, hanno contribuito non poco alla realizzazione di questo lavoro.

Un grazie anche a tutti i membri dei gruppi DELPHI e CMS di Padova, in particolare il prof. Ugo Gasparini, il dott. Ivano Lippi, la dott.ssa Marina Passaseo e il dott. Enrico Piotto.

Infine, ma non per ultimo, ringrazio la mia famiglia, che non mi ha fatto mai mancare il suo sostegno.

Appendice A

Glossario

analisi uno dei tipi di processing eseguiti sugli eventi, in cui gli eventi reali ricostruiti e gli eventi simulati ricostruiti vengono elaborati, per poi verificare differenze e/o analogie tra i due insiemi di dati;

architecture database il database globale in cui ogni elemento rappresenta una singola architettura;

architettura definisce la compatibilità binaria per i file eseguibili: se due computer hanno la stessa architettura, è possibile utilizzare gli stessi file eseguibili per i due sistemi;

capacità (di un tipo di nastro) la massima quantità di informazione che è possibile memorizzare su un nastro del modello considerato;

computational request la richiesta, specificata da un utente del sistema DEMI, di elaborare specifici dati, usando una produzione precedentemente definita;

cost of the task il valore numerico che rappresenta il costo per l'esecuzione del tipo di task considerato, per una particolare architettura (questo parametro viene usato per valutare le diverse performance dei computer delle varie architetture, nell'esecuzione di un task);

data box di una produzione un input/output di un task componente di una produzione. I data box possono essere classificati in base al loro "ruolo" (input, output, per dati temporanei), e per il loro tipo (stream, singleton).

data database il database utilizzato per la memorizzazione delle informazioni relative ai dati gestiti dal sistema DEMI. È strutturato in un global data database globale e in local data database (uno in ogni sito);

data manager il modulo del sistema DEMI che gestisce le operazioni che coinvolgono i file di dati;

data mover la "entità" che deve decidere se, quando e come (usando la Wide Area Network, usando spedizioni via corriere, ecc...) i dati devono essere spostati da un sito ad un altro, per la loro elaborazione;

dati di tipo singleton vengono rappresentati da singoli file;

dati di tipo stream vengono rappresentati da sequenze di file "indipendenti", che vengono elaborati (se input) o prodotti (se output) uno alla volta;

dati reali i dati acquisiti direttamente dagli apparati sperimentali;

dati simulati i dati prodotti attraverso un programma di simulazione (Monte Carlo);

DEMI Global Control System il modulo del sistema DEMI che gestisce le elaborazioni del sistema DEMI a livello globale;

DEMI Local Control System il modulo del sistema DEMI che gestisce le elaborazioni del sistema DEMI a livello di un singolo sito;

disk buffer dischi, o "parti di dischi", utilizzati dal sistema DEMI per la memorizzazione dei file;

disk manager il modulo del data manager che gestisce le operazioni sui file di dati memorizzati su disk buffer;

elementary processing l'elaborazione di una elementary request;

elementary request una computational request sottomessa da un utente viene "scomposta" in elementary request: ogni elementary request corrisponde ad un singolo stream input file, per l'input di tipo stream della corrispondente production. Se non esiste un input di tipo stream, ogni elementary request corrisponde ad una iterazione;

elementary request decomposer il modulo del sistema DEMI che scompone una singola elementary request nei singoli task che compongono la corrispondente produzione;

elementary request identifier permette di identificare senza ambiguità una elementary request (e il relativo processing);

event data sono rappresentati da sequenze di eventi indipendenti. Gli event data corrispondono ai dati di tipo stream;

evento l'unità fondamentale di informazione in ogni esperimento HEP. Un evento è fondamentalmente la registrazione in forma digitale delle tracce lasciate dalle particelle dopo una collisione tra due o più particelle. I vari eventi sono indipendenti tra loro: questo significa che è possibile elaborare un evento senza preoccuparsi dell'elaborazione degli altri eventi;

file identifier permette di identificare senza ambiguità un file di dati, definito nel data database;

global administrator la persona responsabile della gestione del sistema globale. Suoi compiti sono la definizione dell'ambiente, la gestione dei database "centrali", ecc...;

global administrator database il database usato per la memorizzazione delle informazioni relative ai global administrator;

global data database il database globale, in cui ogni elemento rappresenta un file di dati gestito dal sistema DEMI;

global interface il programma utilizzato dal global administrator per "interagire" con il sistema DEMI;

global logbook il logbook in cui vengono memorizzate le informazioni riguardo gli eventi significativi e gli errori occorsi nel sistema DEMI a livello globale;

global processing database il database globale, in cui ogni elemento rappresenta un elementary processing;

global queue of elementary requests la struttura utilizzata per la memorizzazione delle elementary request, che devono essere analizzate dal global scheduler;

global scheduler la "entità" responsabile del mapping delle elementary request ai siti del sistema DEMI, per la loro elaborazione;

job checker il modulo del sistema DEMI che si "interfaccia" con il batch system, per verificare lo stato dell'elaborazione dei job DEMI;

job manager il modulo del sistema DEMI che gestisce gli eventi rilevati dal job checker;

local administrator la persona responsabile della gestione di un sito del sistema DEMI. Suoi compiti sono la definizione dell'ambiente locale, la gestione dei database implementati in quel sito, ecc...;

local buffer database il database in cui vengono memorizzate le informazioni riguardanti i disk buffer del sito locale, utilizzati dal sistema DEMI. Vi è un local buffer database in ogni sito;

local computer database il database in cui vengono memorizzate le informazioni riguardanti i computer presenti nel sito locale, utilizzati dal sistema DEMI. Vi è un local computer database in ogni sito;

local data database il database in cui ogni elemento rappresenta un file di dati fisicamente presente (su disco e/o su nastro) nel sito locale. C'è un local data database in ogni sito;

local data mover il modulo del data manager che gestisce la copia di file da disk buffer a nastri, e viceversa;

local interface il programma utilizzato dagli utenti di un sito (local administrator compresi) per "interagire" con il sistema DEMI;

local logbook il logbook in cui vengono memorizzate le informazioni riguardo gli eventi significativi e gli errori occorsi nel sito locale. Esiste un local logbook per ogni sito del sistema DEMI;

local processing database il database in cui ogni elemento rappresenta l'elaborazione di una elementary request effettuata nel sito locale. C'è un local processing database in ogni sito;

local queue of elementary requests la struttura, implementata in ogni sito, per la memorizzazione delle elementary request che devono essere elaborate nel sito locale;

local scheduler la "entità" responsabile del mapping dei vari task componenti le elementary request ai computer del sito locale, per la loro elaborazione. Vi è un local scheduler in ogni sito del sistema DEMI;

local tape database il database in cui ogni elemento rappresenta un nastro che contiene file di dati. C'è un local tape database in ogni sito;

local tape device database il database in cui vengono memorizzate le informazioni riguardanti i tape device presenti nel sito locale, utilizzati dal sistema DEMI. Vi è un local tape device database in ogni sito;

long term data dati costanti durante l'elaborazione di tutta la sequenza di eventi considerati (es. costanti di calibrazione). I long term data corrispondono ai dati di tipo singleton;

metasistema un sistema di calcolo distribuito, costituito da gruppi eterogenei di computer autonomi, collegati tra loro da una rete locale. Ogni sito del sistema DEMI rappresenta un metasistema;

monte carlo altro nome per 'simulazione';

operatore la persona responsabile del mount dei nastri sugli appositi tape device;

performances il parametro del local computer database che definisce le prestazioni dei computer: più alto è il valore del parametro, migliori sono le performance;

priority level (di una computational request) il parametro numerico che rappresenta il livello di priorità di una computational request. Un valore elevato di questo parametro indica la necessità di soddisfare il prima possibile la richiesta;

processing database il database utilizzato per la memorizzazione delle informazioni riguardanti le elaborazioni eseguite nei vari siti. Il processing database è strutturato in un global processing database, e in local processing database (uno in ogni sito);

production un insieme di task, in relazione tra loro, che permettono di gestire nella sua complessità il problema considerato;

production database il database globale, in cui ogni elemento rappresenta una production, definita da un utente del sistema DEMI;

produzione altro nome per 'production';

raw data la rappresentazione dei dati, reali o simulati, nel formato specifico al detector;

reprocessing l'attività per cui si rielaborano, utilizzando una nuova versione di un programma di analisi, dati che erano già stati elaborati utilizzando una precedente release dell'applicazione considerata;

REQID (REQuest IDentifier) permette di identificare senza ambiguità una richiesta computazionale sottomessa da un utente;

request database il database globale, in cui ogni elemento rappresenta una computational request, sottomessa da un utente del sistema DEMI;

request decomposer il modulo del sistema DEMI che scompone una singola computational request nelle sue elementary request;

richiesta altro nome per 'computational request';

ricostruzione l'interpretazione dei raw data in grandezze significative dal punto di vista fisico. Durante la ricostruzione vengono anche identificati i tipi di particelle che sono state generate nelle collisioni;

run sequenza composta da centinaia o migliaia di eventi indipendenti. Si considerano elaborazioni sui run (e non sui singoli file) per fare in modo che le operazioni di setup necessarie nei vari processing siano eseguite per ogni run, e non per ogni evento, in modo da ridurre i tempi di esecuzione. Un run corrisponde ad un file di tipo stream;

simulazione una dei tipi di elaborazione sugli eventi. Scopo della simulazione è la generazione di eventi simulati, che saranno poi analizzati per un "confronto" con i corrispondenti dati reali. I dati reali e i dati simulati hanno la stessa struttura, in modo da poter utilizzare gli stessi programmi per la ricostruzione e analisi;

singleton data box di una produzione il data box di una produzione a cui si associa, in una richiesta, un singleton set costituito da un unico singleton file;

singleton input/output file un file, che costituisce un singleton input/output set;

singleton input/output set un insieme costituito da un unico singleton input/output file, che in una richiesta viene associato ad un data box di input/output di tipo singleton della corrispondente produzione;

sito un gruppo di computer, collegati tra loro attraverso una rete locale, utilizzati dal sistema DEMI. Il sistema DEMI e' costituito da vari siti, connessi tra loro attraverso una rete geografica;

sito principale il laboratorio dove sono installati gli apparati sperimentali di un esperimento HEP;

stream data box di una produzione il data box di una produzione a cui si associa, in una richiesta, uno stream set costituito da una collezione di stream file;

stream input/output file un file, elemento di uno stream input/output set;

stream input/output set una collezione di stream input/output file, che in una richiesta vengono associati ad un data box di input/output di tipo stream della corrispondente produzione;

struttura centralizzata il servizio, previsto in alcuni esperimenti HEP, che si occupa di gestire le elaborazioni che devono essere eseguite su grossi volumi di dati, che richiedono quindi un'elevata potenza computazionale. La struttura centralizzata riceve le richieste dagli utenti, schedula e sottomette i relativi job sui calcolatori disponibili;

tape device il dispositivo utilizzato per la lettura/scrittura di dati da/su nastro;

tape device model un modello di tape device, e dei relativi nastri (es. DLT, Exabyte 8 mm, DAT, ecc...);

tape device model database il database globale in cui ogni elemento rappresenta un tape device model;

tape identifier permette di identificare senza ambiguità un nastro, definito nel local tape database;

tape manager il modulo del data manager che gestisce le operazioni sui file di dati memorizzati su nastri;

task un'attività elementare di elaborazione, che gestisce un particolare aspetto del problema globale. I task sono gli elementi che compongono le production;

task database il database globale, in cui ogni elemento rappresenta un tipo di task, definito da un utente del sistema DEMI;

task starter il modulo del sistema DEMI che verifica quali task possono essere eseguiti;

user database il database utilizzato per la memorizzazione dei dati riguardanti gli utenti del sito locale. Vi è uno user database in ogni sito;

user identifier permette di identificare senza ambiguità un utente, definito nello user database o nel global administrator database;

utente generico (general user) la persona che ha privilegi di default nell'uso del sistema DEMI: può definire task, production, può sottomettere richieste computazionali, ecc...

Appendice B

Ambienti per la programmazione parallela

La programmazione di applicazioni parallele è un compito molto complesso, in quanto è necessario gestire direttamente molti dettagli implementativi: la sincronizzazione e la comunicazione tra i vari processi che costituiscono il programma, il mapping dei processi ai processori disponibili, ecc ...

Considerando un tipo di parallelismo a granularità di tipo coarse grain (il tipo di parallelismo da considerare nel sistema da realizzare), la progettazione di un'applicazione parallela può essere divisa in due fasi [18]:

1. la definizione delle diverse componenti sequenziali;
2. la definizione delle relazioni esistenti tra questi elementi.

Per semplificare lo sviluppo di applicazioni parallele di questo tipo, negli ultimi anni sono stati sviluppati appositi ambienti visuali. Alcuni di essi vengono esaminati nei prossimi paragrafi.

B.1 Grafi di computazione

Il formalismo del **grafo di computazione** [17] è uno strumento molto utile per la definizione di un'applicazione parallela, ed è utilizzato in molti ambienti: un grafo di computazione è un grafo diretto, in cui i nodi rappresentano componenti sequenziali, mentre gli archi definiscono relazioni tra queste componenti. Una relazione tra due nodi può definire uno scambio di messaggi, o una sincronizzazione (ossia un vincolo di precedenza) tra questi due nodi.
I grafi di computazione:

- permettono lo sviluppo di applicazioni parallele, secondo la metodologia in due fasi prima descritta (la definizione delle componenti sequenziali, e la definizione delle relazioni tra queste componenti): i nodi del grafo di computazione rappresentano le componenti sequenziali, mentre gli archi vengono utilizzati per definire le relazioni;

- consentono di combinare con facilità componenti sequenziali precedentemente definite;

- permettono di astrarre la rappresentazione del programma dalla sua implementazione;

- favoriscono l'utilizzo della metodologia di *programmazione top-down*: durante lo sviluppo dell'applicazione è possibile sostituire un nodo con un grafo, come rappresentato nell'esempio di figura B.1. In questo esempio, il nodo B viene sostituito con il sottografo {B1, B2, B3, B4}. È importante notare come questa sostituzione non richieda alcuna modifica nei nodi A e C;

- consentono di individuare facilmente eventuali problemi o anomalie nella ripartizione delle funzionalità tra le varie componenti di un'applicazione parallela.

  Considerando ad esempio il grafo di computazione di figura B.2 (a), in cui per ogni nodo è riportata la stima del tempo di esecuzione, se i quattro moduli vengono eseguiti in parallelo su quattro processori, è richiesto un tempo di esecuzione totale pari a 11.

  Si può notare facilmente come l'esecuzione dei primi due nodi richieda un tempo di esecuzione molto superiore (9) a quello richiesto per l'esecuzione degli altri due (1). Questa "anomalia" può suggerire, se possibile, una ristrutturazione del programma, in modo da ottenere un insieme di procedure con tempi di esecuzione paragonabili, come mostrato in figura B.2 (b): in questo caso il carico di lavoro totale rimane invariato (pari a 20), ma il miglior bilanciamento permette di ridurre a 8 il tempo di esecuzione totale.

B.1.1 Esempi di utilizzo dei grafi di computazione

Vengono presentati due esempi, che dimostrano come i grafi di computazione possono essere un utile strumento per definire applicazioni di tipo parallelo.

Nel primo caso viene esaminato un programma parallelo progettato secondo il modello a memoria condivisa, mentre nel secondo esempio viene considerata un'applicazione basata sul modello a scambio di messaggi [18].

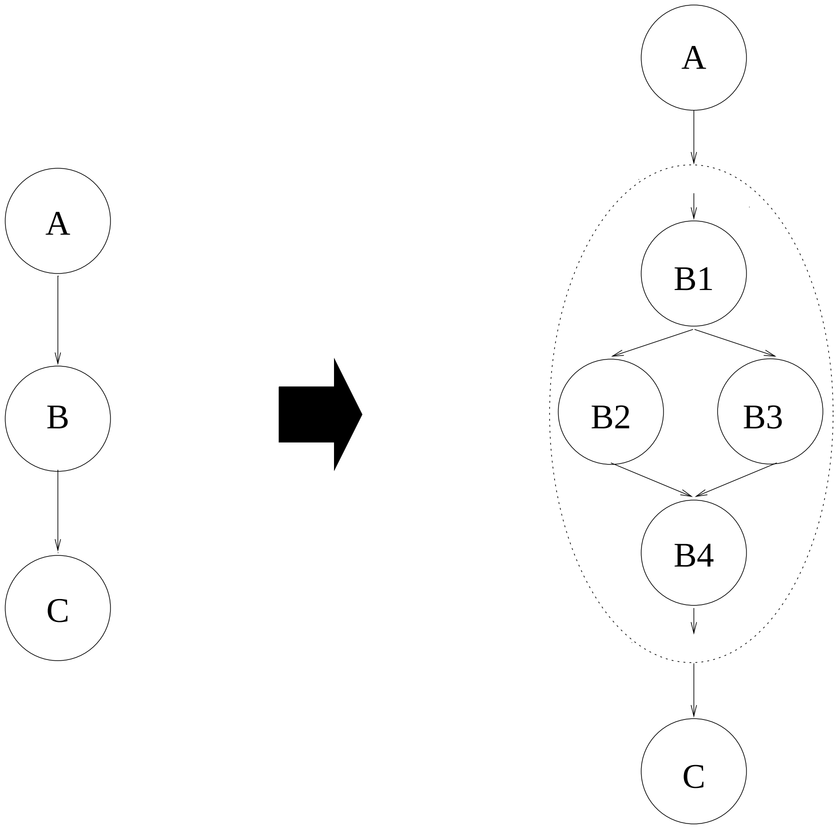


Figura B.1: Esempio di utilizzo dei grafi di computazione per la programmazione top-down

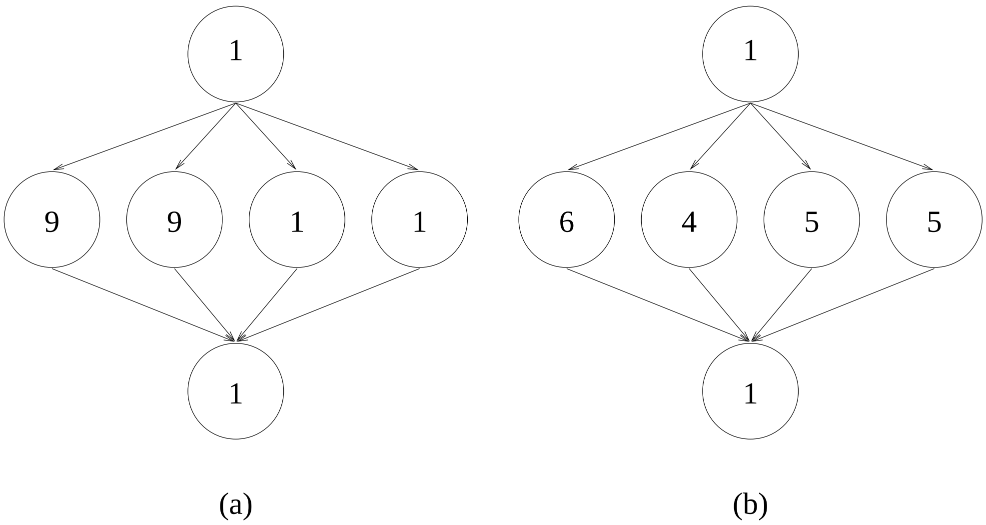


Figura B.2: Due grafi di computazione che rappresentano un'applicazione parallela, rispettivamente prima (a) e dopo (b) una ristrutturazione

Esempio di applicazione parallela basata sul modello a memoria condivisa

Si consideri un programma composto da cinque moduli (*M1*, *M2*, *M3*, *M4*, *M5*), che operano su cinque variabili condivise (x, y, z, v, w):

```
          M1                   M2                   M3
           .                    .                    .
(1)        .         (3)        .         (5)        .
           .                    .                    .

(a)     x:=1         (b)     y:=2         (c)   z:=x+y+3

           .                    .                    .
(2)        .         (4)        .         (6)        .
           .                    .                    .
```

```
            M4                         M5
             .                          .
   (7)       .               (9)        .
             .                          .

   (d)   v:=x+4              (e)  w:=x+z+v+5

             .                          .
   (8)       .               (10)       .
             .                          .
```

Le istruzioni delle porzioni di codice non specificate (rappresentate dai numeri 1-10) non accedono alle cinque variabili condivise.

Per definire il comportamento del programma, anziché elencare tutte le possibili combinazioni descriventi l'ordine in cui le varie istruzioni possono essere eseguite, si può utilizzare un grafo di computazione, come indicato in figura B.3

I nodi del grafo rappresentano istruzioni, o porzioni di codice (il nodo i rappresenta l'istruzione/porzione di codice *(i)*). Gli archi, invece, rappresentano vincoli di precedenza: se esiste un arco $x \to y$, il codice *(x)* dovrà essere eseguito prima del codice *(y)* (ad esempio, con riferimento al caso considerato, l'istruzione *(e)* dovrà essere eseguita dopo le istruzioni *(a)*, *(c)* e *(d)*). Se invece non vi sono archi tra due nodi, le corrispondenti istruzioni/porzioni di codice possono essere eseguite in parallelo (ad esempio, sempre con riferimento al grafo di figura, le istruzioni *(a)* e *(b)* possono essere eseguite in parallelo).

Esempio di applicazione parallela basata sul modello a scambio di messaggi

Si consideri un programma costituito da moduli che invocano routine di una libreria di scambio di messaggi.

Quando un modulo deve inviare un messaggio ad un altro modulo, utilizza una procedura *send()*; al contrario, quando un modulo necessita di un dato da parte di un altro modulo, invoca una procedura *receive()*: l'esecuzione è sospesa finché il dato non viene ricevuto.

Il comportamento di un programma di questo tipo può essere rappresentato da un grafo di computazione, in cui i nodi descrivono i moduli, mentre gli archi rappresentano il flusso dei messaggi.

Nell'esempio di figura B.4 è rappresentata un'applicazione parallela costituita da tre moduli (A, B, C), che comunicano tra loro attraverso messaggi (nel grafo

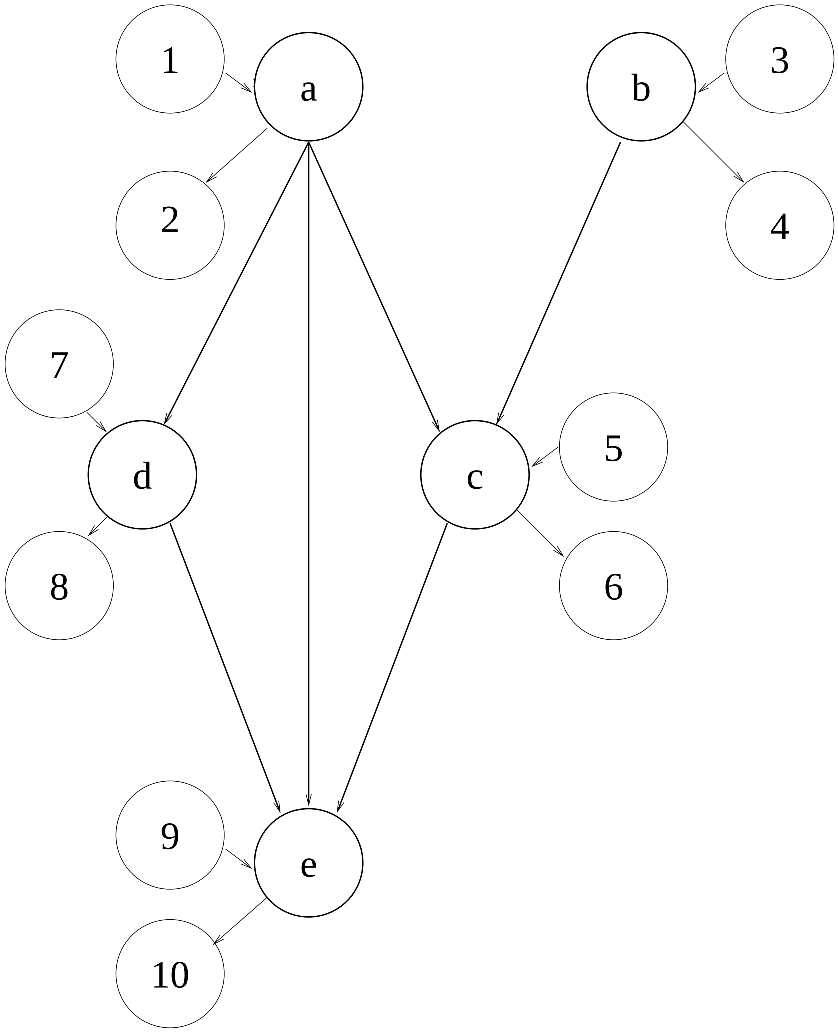


Figura B.3: Grafo di computazione con vincoli di precedenza tra i moduli costituenti il programma parallelo

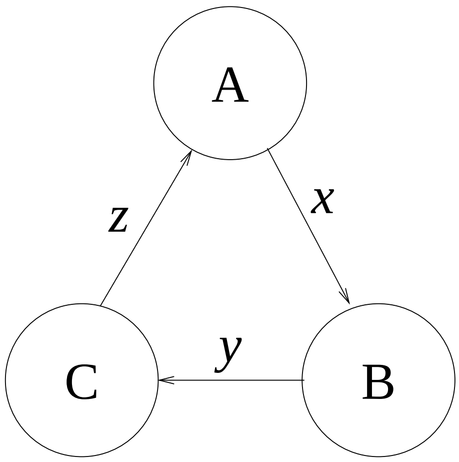


Figura B.4: Grafo di computazione descrivente le comunicazioni tra i moduli di un programma parallelo

gli archi sono etichettati con il valore dei dati che vengono scambiati). Ad esempio l'arco dal nodo A al nodo B, etichettato con il valore x, indica che il modulo A invia al modulo B un messaggio, contenente il dato x.

B.2 Analisi di alcuni ambienti per la programmazione parallela

Nei prossimi paragrafi vengono esaminati alcuni ambienti visuali per la programmazione parallela.
In particolare vengono considerati i sistemi:

- HeNCE
- CODE
- VPE
- Enterprise
- Paralex
- Meander

Per ognuno di questi ambienti vengono analizzate le specifiche caratteristiche e funzionalità, con particolare attenzione alle esigenze per il sistema DEMI.

B.3 HeNCE

HeNCE (Heterogeneous Network Computing Environment) [13] [14] [18] è un ambiente grafico per la programmazione di applicazioni parallele.
Utilizzando questo strumento, il programmatore, attraverso un'apposita interfaccia grafica, può:

- definire un'applicazione parallela;
- compilarla;
- eseguirla;
- farne il tracing.

I programmi sviluppati attraverso HeNCE possono essere eseguiti in un ambiente eterogeneo di workstation Unix, che utilizzano PVM [25] come sistema di comunicazione a scambio di messaggi.

B.3.1 Definizione di un'applicazione

In HeNCE un programma viene definito attraverso un grafo di computazione, in cui i nodi rappresentano procedure (scritte in linguaggio C o Fortran), mentre gli archi definiscono vincoli di precedenza: un arco da un nodo A ad un nodo B indica quindi che il codice associato al nodo A deve essere eseguito prima del codice associato al nodo B.

In figura B.5 è mostrato il tool grafico che viene utilizzato per la specifica del grafo di computazione (il grafo può comunque essere definito anche attraverso un'interfaccia di tipo testuale).
Nella definizione di un grafo di computazione si possono utilizzare sei tipi di costrutti:

- Nodi **subroutine**

  Ad un nodo subroutine è associata una procedura, una lista di parametri di input, e una lista di parametri di output. Quando un nodo subroutine deve essere eseguito, viene invocata la procedura ad esso associato.

  In HeNCE un nodo subroutine può comunicare con i nodi adiacenti solo all'inizio e alla fine della propria esecuzione: le liste dei parametri di input e di output definiscono quali dati devono essere scambiati in questa comunicazione. Non è invece ammesso l'utilizzo di variabili globali.

  Ad un nodo subroutine possono essere associate più implementazioni, una per ogni architettura presente nel sistema eterogeneo considerato.

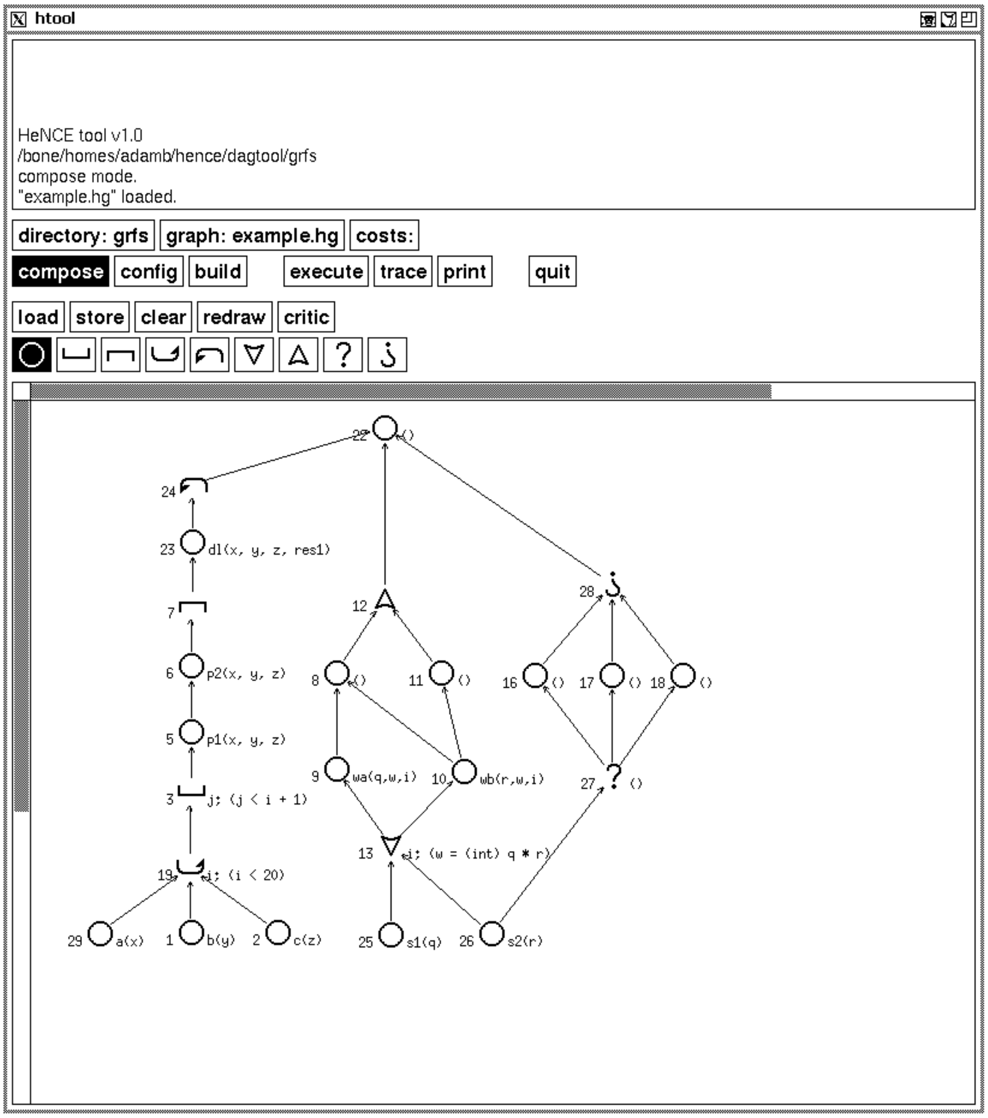


Figura B.5: Tool HeNCE per la definizione di un'applicazione parallela

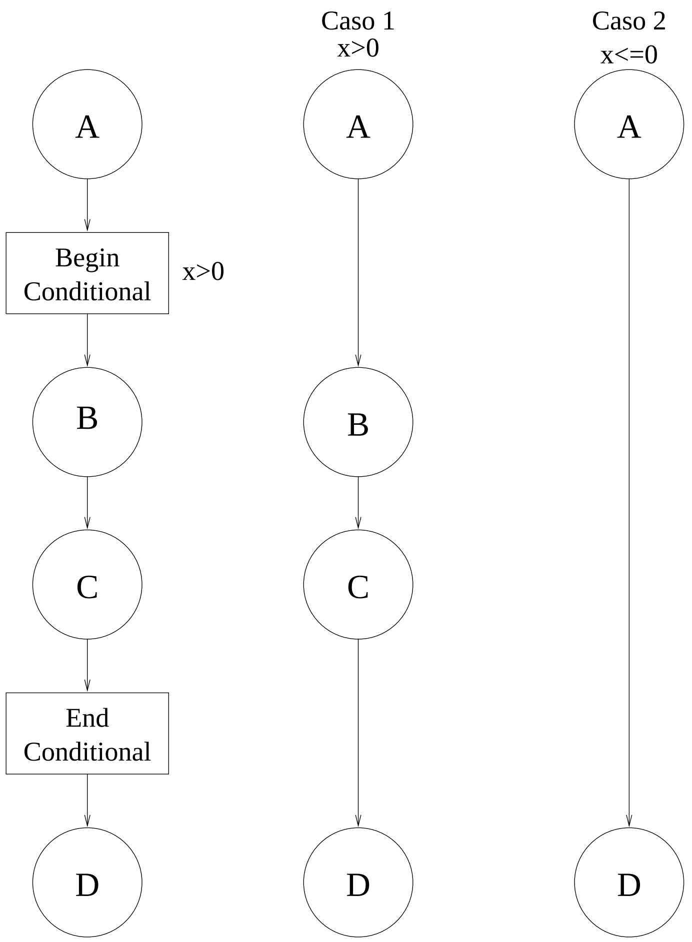


Figura B.6: Esempio di utilizzo del costrutto conditional

- Archi **single dependency**

  Un arco dependency collega due nodi subroutine, definendo un vincolo di precedenza tra questi due nodi.

- Costrutti **conditional**

  Usando il costrutto conditional, è possibile definire un sottografo che deve essere eseguito solo in certe condizioni: ad un nodo *Begin Conditional* viene associata un'espressione booleana (v. figura B.6); se questa è vera, allora viene eseguito il codice associato al sottografo compreso tra i nodi *Begin Conditional* e *End Conditional*. Se invece l'espressione booleana è falsa, tale codice non viene eseguito: l'esecuzione riprende dal nodo successivo al nodo *End Conditional*.

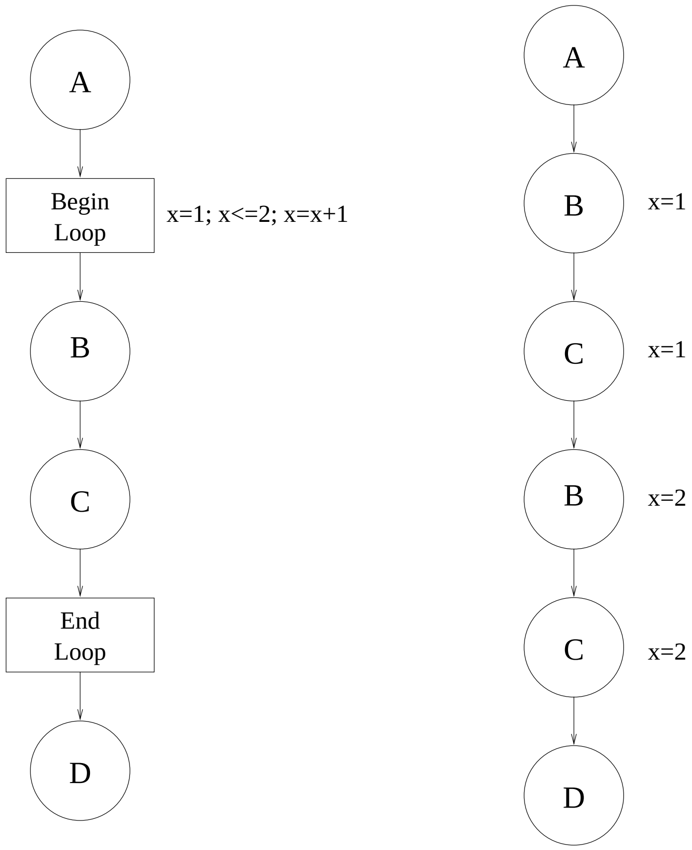


Figura B.7: Esempio di utilizzo del costrutto loop

- Costrutti **loop**

  Attraverso questo costrutto si esegue ripetutamente (un numero di volte pari al valore di un'espressione) il codice associato ai nodi di un sottografo, che costituisce il corpo del ciclo.

  La sintassi e la semantica sono del tutto analoghe a quelle del costrutto *for* del linguaggio C: ad un particolare nodo *Begin Loop* viene associato il nome di una variabile indice, il suo valore iniziale, l'istruzione che ne modifica il valore, e la condizione di terminazione. Il codice associato al sottografo delimitato dai nodi *Begin Loop* e *End Loop* viene eseguito finché la condizione di terminazione viene soddisfatta (dopo ogni iterazione viene eseguita l'istruzione che modifica il valore della variabile indice).

  Nell'esempio di figura B.7, il corpo del ciclo deve essere eseguito due volte.

- Costrutti **fan**

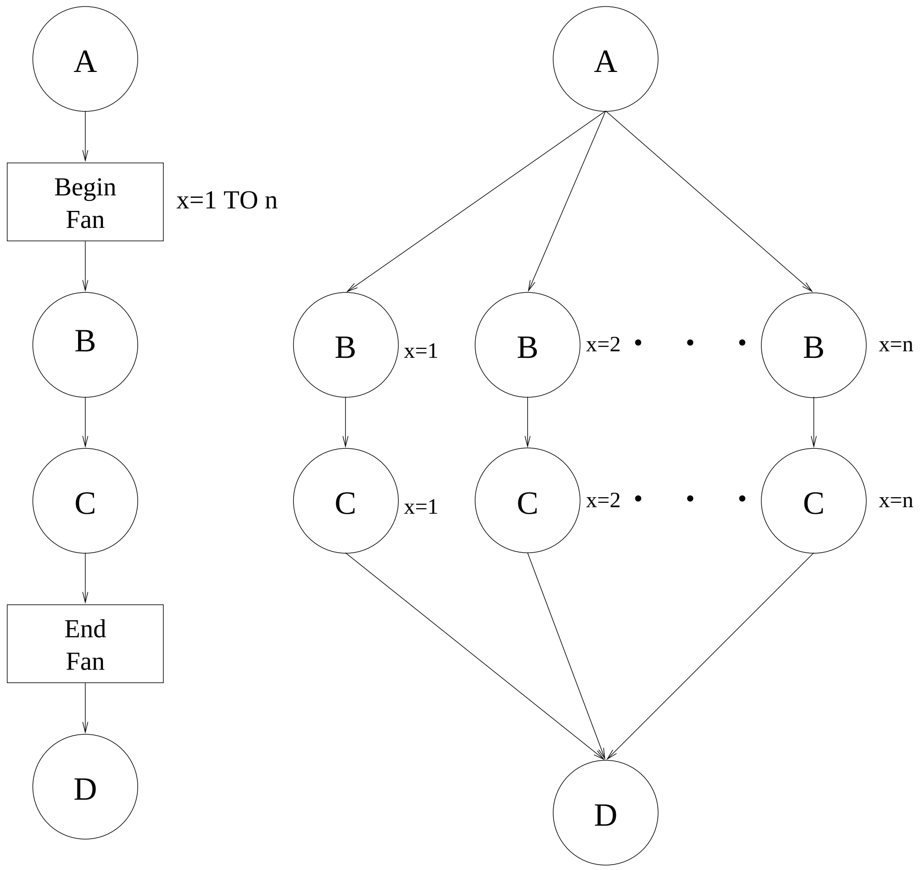


Figura B.8: Esempio di utilizzo del costrutto fan

Il costrutto fan permette di replicare un sottografo, e di eseguirne in parallelo il codice associato.

Ad un nodo *Begin Fan* (v. figura B.8) viene associata un'espressione del tipo:

```
var = exp1 TO exp2
```

dove *var* è il nome di una variabile, e *exp1* e *exp2* sono due espressioni numeriche. Il sottografo, delimitato dai nodi *Begin Fan* e *End Fan*, viene replicato un numero di volte pari a *exp2 - exp1 + 1*.

- Costrutti **pipe**

  Ad un particolare nodo *Begin Pipe* (v. figura B.9) viene associata un'espressione del tipo:

```
var = exp1 TO exp2
```

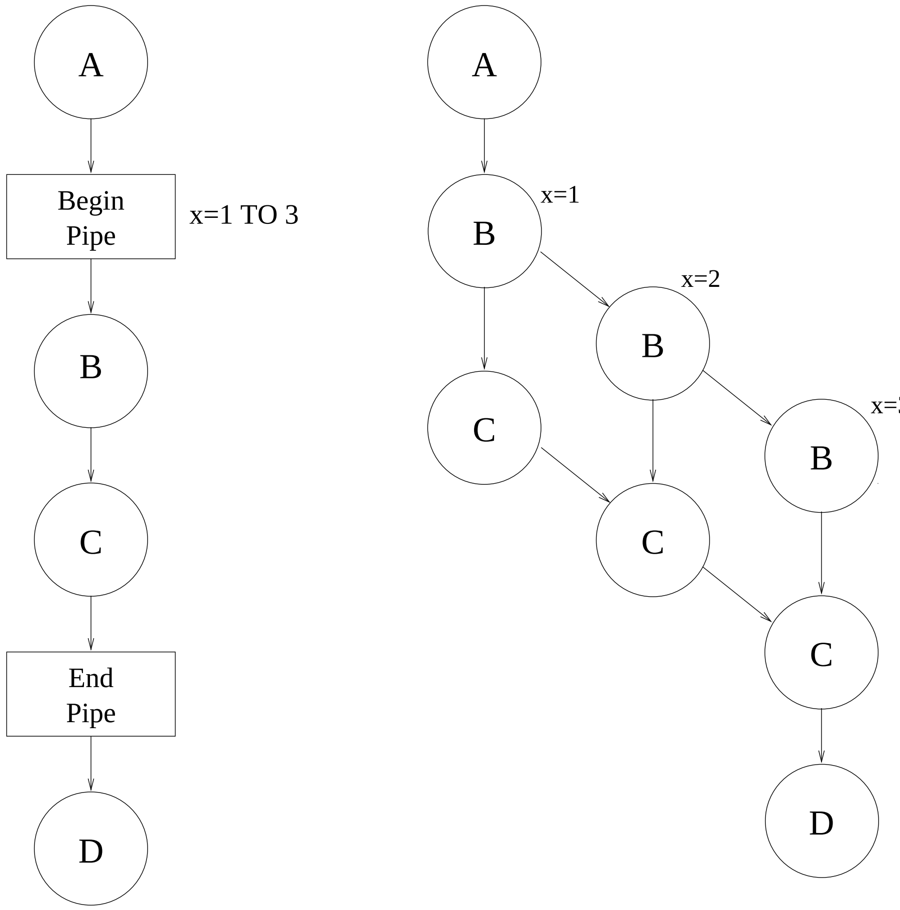


Figura B.9: Esempio di utilizzo del costrutto pipe

dove *var* è il nome di una variabile, e *exp1* e *exp2* sono due espressioni numeriche. Il sottografo delimitato dai nodi *Begin Pipe* e *End Pipe* viene replicato durante l'esecuzione un numero di volte pari a *exp2 - exp1 + 1*.

Con riferimento all'esempio di figura B.9, per ogni valore assunto dalla variabile x, vengono create e attivate nuove istanze dei nodi B e C, e vengono distrutte quelle precedenti.

B.3.2 Compilazione

Una volta che il grafo di computazione per l'applicazione parallela è stato definito, ed è stato specificato il codice associato ai vari nodi subroutine, il sistema "integra" tale codice con le opportune chiamate alle routine PVM.

Si procede poi con la compilazione, per ogni architettura presente nel sistema considerato.

B.3.3 Esecuzione

Dopo la compilazione, il programma HeNCE può essere eseguito sulla macchina virtuale definita dall'utente.

L'esecuzione dell'applicazione ha inizio dalla procedura associata al nodo iniziale del grafo, e termina quando sono state eseguite tutte le procedure associate ai nodi raggiungibili da questo. Una procedura, che corrisponde ad un certo nodo del grafo di computazione, può essere eseguita solo quando la procedura associata al nodo "predecessore" ha terminato la sua esecuzione.

Il programmatore ha la possibilità di definire la cosiddetta *matrice dei costi*, in cui specifica il tempo stimato per l'esecuzione delle varie procedure sui vari calcolatori costituenti la macchina virtuale: l'elemento *(m,p)* di questa matrice è un numero che rappresenta la stima del tempo di esecuzione per la procedura p sulla macchina m.

La matrice dei costi viene usata dall'algoritmo di mapping, che deve associare le varie procedure ai processori disponibili: indicando con $load_m$ il load del processore m, e con $cost_{m,p}$ l'elemento *(m,p)* della matrice dei costi, la procedura p viene "associata" al processore m per cui è minimo il valore della somma $load_m + cost_{m,p}$.

In questo algoritmo viene considerato solo il load dovuto a programmi HeNCE: non si considerano eventuali altri job di utenti generici.

B.3.4 Tracing

L'interfaccia HeNCE comprende un modulo per la visualizzazione dell'esecuzione di un programma. In figura B.10 è mostrato un esempio.

Attraverso questo strumento è possibile verificare come le varie procedure sono state "mappate" ai processori disponibili, si può controllare lo stato dei vari componenti della macchina virtuale (compresa l'attività dei processori), ecc...

B.4 CODE

CODE [18] [39] [41] è un ambiente visuale per lo sviluppo di applicazioni parallele, eseguibili su un'unica architettura.
Usando questo ambiente è possibile:

- definire applicazioni parallele;

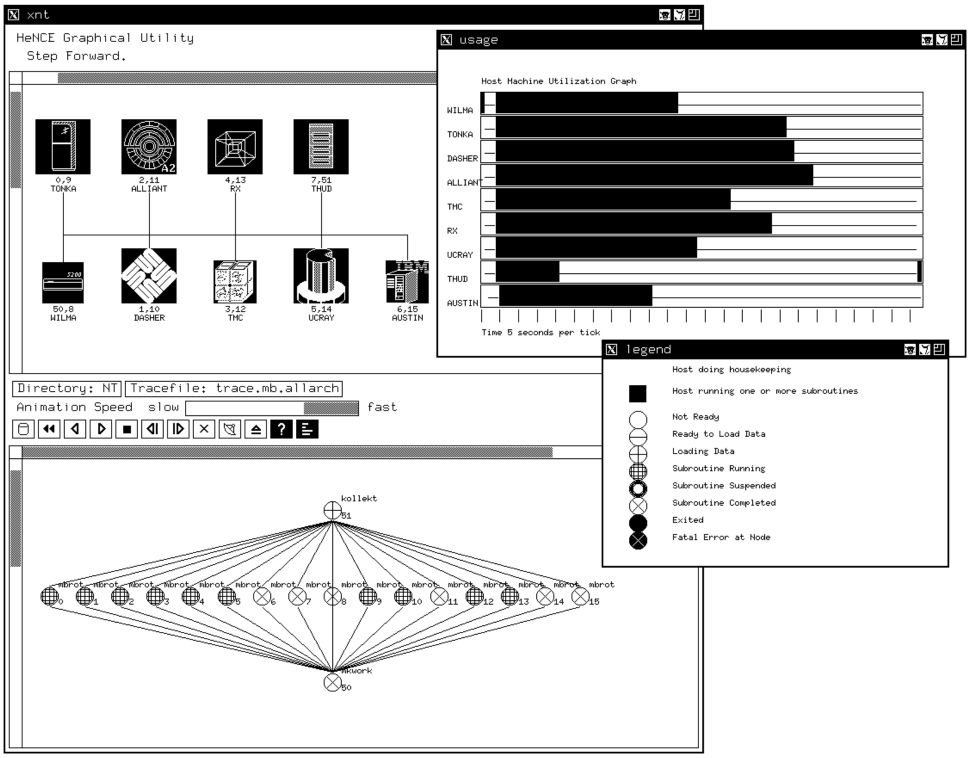


Figura B.10: Esempio di tracing di un'applicazione con HeNCE

- compilarle;
- eseguirle.

Non sono invece disponibili funzionalità di tracing o debugging.
In CODE viene usato PVM [25] come sistema di message passing.

B.4.1 Definizione di un'applicazione

Un'applicazione parallela viene definita in CODE attraverso un grafo, in cui i nodi rappresentano unità computazionali o variabili condivise, mentre gli archi rappresentano "flussi" di informazioni tra i nodi. Ogni arco collega due porte, ognuna delle quali è associata ad un nodo del grafo.
Esistono diversi tipi di nodi con i quali si può costruire il grafo:

- Nodi **Unit Computation** (**UC**)

  Un nodo di questo tipo rappresenta un'elaborazione sequenziale: ad esso è associata una procedura (scritta in linguaggio C).

  Ad un nodo UC sono poi associati dati locali, *regole di firing*, e *regole di routing*. Le regole di firing definiscono le condizioni che devono essere soddisfatte affinché la procedura possa essere eseguita. Le regole di routing, invece, definiscono in che modo il risultato della computazione viene inviato ai vari nodi.

  In figura B.11 è mostrato un esempio di specifica delle regole di firing e routing per un nodo UC: in questo caso le regole di firing stabiliscono che l'esecuzione deve iniziare solo quando, sulle porte A e B sono disponibili dei valori, che vengono assegnati rispettivamente alle variabili X e Y. La regola di routing, invece, definisce che, se l'espressione booleana $X > 1$ *and* $Y < 2$ è vera, allora il valore $X+Y$ deve essere assegnato alla porta C.

- Nodi **Name sharing**

  Un nodo di questo tipo rappresenta un insieme di variabili che possono essere condivise tra le procedure definite nel programma. Questo nodo viene poi collegato con archi ai nodi che utilizzano questi dati comuni (CODE gestisce direttamente la mutua esclusione per l'accesso a queste variabili).

- Nodi **Call** e nodi **Input/Output interface**

  Un nodo call viene utilizzato quando un grafo deve "chiamare" un altro grafo, in modo da collegare le corrispondenti computazioni.

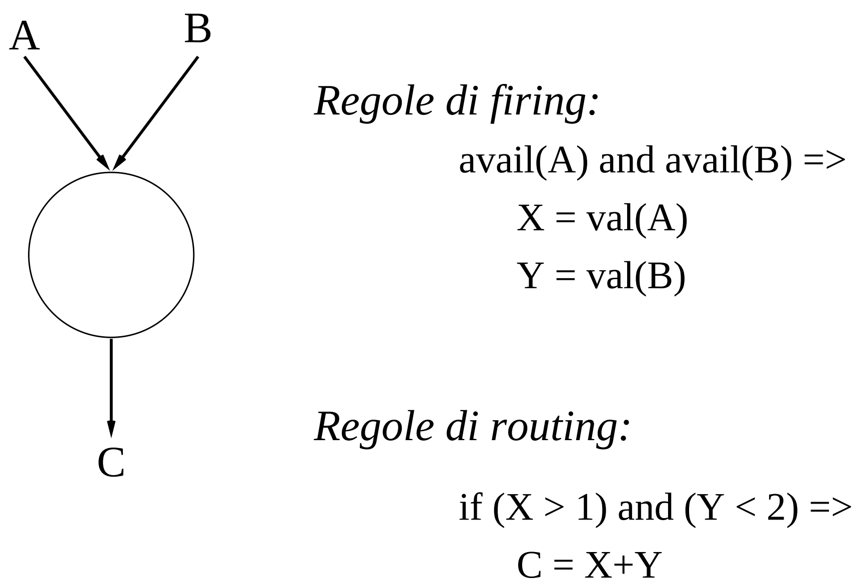


Figura B.11: Esempio di nodo Unit Computation, con le associate regole di firing e di routing

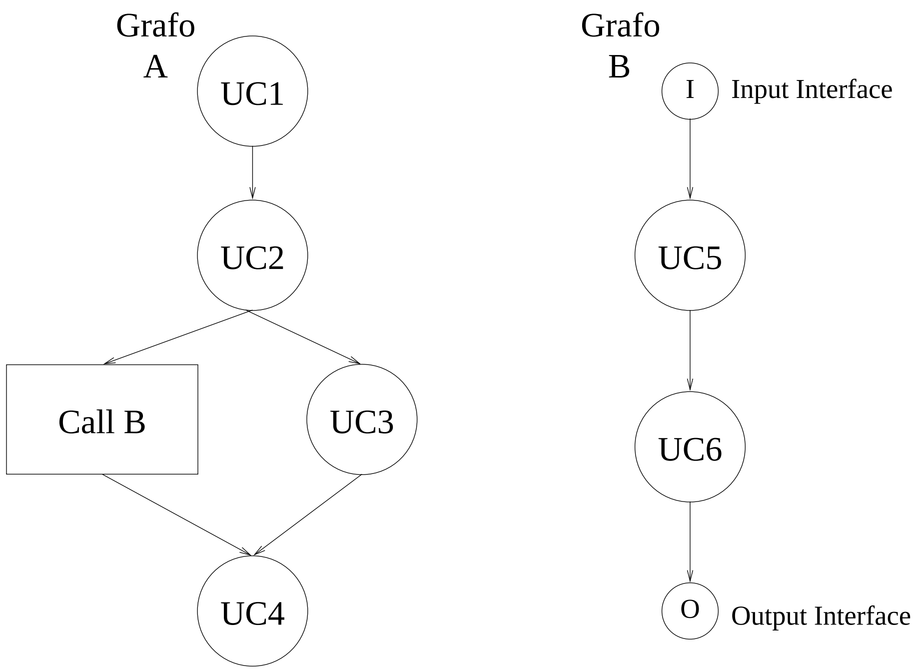


Figura B.12: Esempio di utilizzo dei nodi Call e dei nodi Interface

Nell'esempio di figura B.12 il grafo A contiene un nodo call, che "chiama" il grafo B.

Nel grafo chiamato sono presenti un nodo input interface e un nodo output interface (i nodi I e O del grafo B), che gestiscono il passaggio dei parametri tra grafo chiamante e grafo chiamato.

Quando un grafo "invoca" un altro grafo, l'esecuzione nel grafo chiamante prosegue con il nodo UC successivo, senza dover attendere che l'elaborazione delle procedure associate al grafo chiamato sia stata completata. In questo modo è possibile avere in esecuzione più procedure allo stesso tempo.

- Nodi **Creation**

  Un nodo creation è un particolare nodo di tipo input interface, tale che un parametro associato ad un nodo di questo tipo è visibile (ma non modificabile) da tutti i nodi del grafo chiamato (un parametro associato ad un nodo input interface, invece, è visibile e modificabile solo dal nodo UC che lo riceve).

Nel modello CODE un'applicazione parallela può quindi essere rappresentata da più grafi; tra questi ne è presente uno particolare, chiamato *main*, che contiene il nodo UC associato alla procedura che deve essere eseguita per prima. Il grafo main può poi contenere "chiamate" agli altri grafi.

B.4.2 Compilazione ed esecuzione

Dopo che il grafo è stato definito, CODE integra il codice specificato dall'utente con le chiamate alle apposite procedure della libreria PVM, che si occupano di gestire la comunicazione tra i processi.

Il programma viene poi compilato sul calcolatore considerato, e mandato in esecuzione.

B.5 VPE

VPE (Visual Programming Environment) [40] è un ambiente visuale per la programmazione di applicazioni parallele, basate sul modello a scambio di messaggi.

VPE può essere utilizzato in un ambiente eterogeneo, costituito da un insieme di workstation che utilizzano il sistema a scambio di messaggi PVM [25] o MPI [54].

Usando VPE il programmatore ha la possibilità di:

- definire un'applicazione parallela;
- compilarla;
- eseguirla.

Per le future implementazioni di VPE è prevista la realizzazione di una funzionalità di tracing, attraverso la quale l'utente avrà la possibilità di controllare direttamente lo stato di esecuzione dell'applicazione.

B.5.1 Definizione di un'applicazione

Per specificare un'applicazione parallela, può essere usato un apposito tool grafico: l'applicazione viene definita attraverso un insieme di grafi.

I nodi di questi grafi rappresentano processi, mentre gli archi definiscono "canali" attraverso cui passano i messaggi inviati da un processo ad un altro. Ogni arco connette due porte, ognuna associata ad un nodo del grafo.

Per costruire il grafo che definisce l'applicazione, si possono utilizzare quattro tipi di costrutti:

- Nodi **compute** (**comp**)

  Un nodo compute rappresenta un processo.

  Ad un nodo di questo tipo è associato codice (scritto in linguaggio C o Fortran) che può contenere chiamate a routine VPE per lo scambio di messaggi. Il programmatore può indicare uno specifico computer (o un tipo di computer) per l'esecuzione di questo codice.

  Ogni nodo compute può essere replicato: al momento dell'esecuzione saranno attivate diverse istanze, ognuna operante su propri dati.

- **Porte di input e di output**

  Ad ogni nodo compute possono essere associate porte di input e/o porte di output, usate per lo scambio di messaggi con gli altri processi: quando si deve inviare/ricevere un messaggio, viene invocata un'apposita routine VPE, in cui si specifica il nome della porta di output/input da usare.

- Nodi **call** e **interface**

  Un'applicazione parallela può essere definita attraverso più grafi collegati tra loro. Il nodo call viene utilizzato per rappresentare la "chiamata" ad un grafo, da parte di un altro grafo.

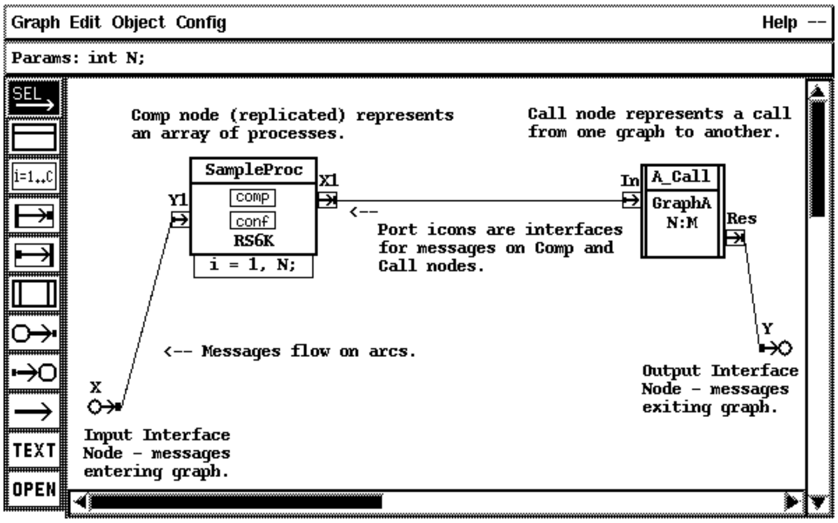


Figura B.13: Esempio di grafo VPE

- Nodi **interface**

  Il grafo "chiamato" attraverso un nodo call deve contenere due nodi particolari, chiamati *nodo input interface*, e *nodo output interface*: la loro funzione è quella di gestire il passaggio dei parametri tra il grafo chiamante e il grafo chiamato.

In figura B.13 viene mostrato un esempio di grafo VPE, in cui compaiono i quattro costrutti descritti.

A differenza di HeNCE e CODE, in cui la comunicazione tra i diversi moduli può avvenire solo all'inizio o alla fine della propria esecuzione, in VPE ogni modulo può interagire con gli altri moduli in qualsiasi momento.

B.5.2 Compilazione ed esecuzione

Una volta che l'applicazione parallela è stata definita, il sistema sostituisce le chiamate alle routine VPE con chiamate alle procedure della libreria di comunicazione utilizzata (PVM o MPI).

Il programma viene poi compilato ed eseguito.

Per quel che riguarda il problema del mapping, ogni modulo viene "associato" al calcolatore specificato dal programmatore durante la definizione dell'applicazione; nel caso in cui sia stato definito solamente il tipo di computer da utilizzare, viene scelta la "prima" macchina disponibile di quell'architettura.

B.6 Enterprise

Enterprise [52] è un ambiente visuale che permette:

- la definizione
- la codifica
- l'esecuzione
- il debugging
- il tracing

di applicazioni che devono essere eseguite in un ambiente distribuito, costituito da un insieme di workstation Unix.

Enterprise può utilizzare come sistema di message-passing PVM [25], ISIS [15], o NMP [35].

B.6.1 Definizione di un'applicazione parallela

Come visto, la definizione di un'applicazione parallela può essere divisa in due fasi: la definizione delle componenti sequenziali, e la specifica delle relazioni tra questi elementi. Con riferimento a questo modello, Enterprise usa una metodologia di programmazione basata sull'uso di due componenti:

- Il **Programming Model**

  Permette di definire il codice dei vari moduli costituenti l'applicazione parallela.

- Il **Metaprogramming Model**

  Permette di specificare le relazioni che esistono tra i vari moduli, attraverso la definizione del cosiddetto *asset graph*.

Il programmatore, nella definizione dell'applicazione parallela, ha quindi la possibilità di operare in due possibili modi:

1. disegnare l'asset graph, e poi definire il codice associato alle varie procedure;
2. scrivere il codice associato alle procedure, e in un secondo tempo definire l'asset graph.

Programming model

Nel programming model un'applicazione parallela viene definita attraverso un insieme di moduli, che interagiscono tra loro.

Ogni modulo è costituito da una *entry procedure*, che può essere invocata dagli altri moduli, e da una serie di *internal procedure*, che possono essere chiamate solo dalle altre procedure definite nello stesso modulo. L'entry procedure e le internal procedure vengono scritte dal programmatore in linguaggio C. Ogni modulo può accedere solo alle variabili locali: non è ammesso l'uso di variabili globali.

Utilizzando esclusivamente il programming model, l'unico strumento per definire le interazioni tra le diverse componenti è l'uso delle chiamate ai moduli. Considerando ad esempio un modulo A che invoca durante la sua esecuzione un modulo B, si possono presentare due possibili casi:

1. Il modulo chiamato B deve restituire un valore al modulo chiamante A (*f-call*).

   In questo caso A blocca la sua esecuzione, in attesa che B restituisca il dato.

2. Il modulo chiamato B non deve restiture un valore al modulo chiamante A (*p-call*).

   In questo caso A non deve attendere il completamento dell'esecuzione del modulo B: i due moduli possono essere eseguiti in parallelo (*parallelismo puramente asincrono*).

Metaprogramming model

Nel metaprogramming model il programmatore definisce la struttura del programma, costruendo l'asset graph.

La definizione dell'asset graph avviene selezionando alcuni elementi, detti *asset*, che ne costituiscono i nodi. Un asset può anche essere replicato, in modo che vi sia più di un processo che ne esegue il codice associato: in questo caso il programmatore deve specificare il fattore di replicazione, ovvero il numero di istanze che devono essere eseguite.

Esistono vari tipi di asset:

- **Enterprise**

  L'applicazione parallela è all'inizio rappresentata da un singolo asset di tipo enterprise. Durante la fase di sviluppo, questo viene "espanso" con altri asset, come rappresentato nell'esempio di figura B.14.

  Un asset enterprise non può essere replicato.

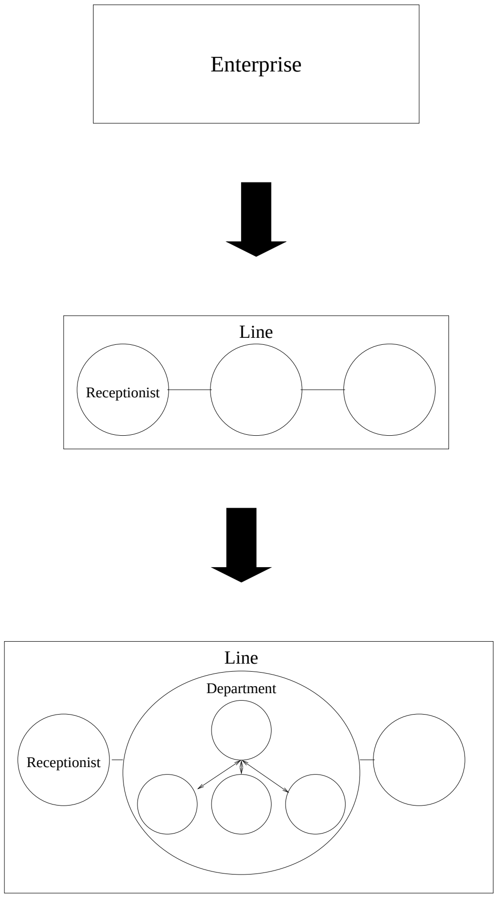


Figura B.14: Esempio di “espansione” di un asset enterprise

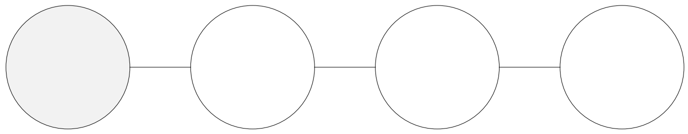


Figura B.15: Esempio di utilizzo dell'asset line

- **Individual**

  Un asset individual rappresenta un singolo modulo di tipo sequenziale. Ad esso è associata una procedura, che viene eseguita quando viene invocata da una procedura di un altro asset di tipo individual.

  Un asset individual non può contenere altri asset.

- **Line**

  Un asset di tipo line è costituito da un insieme di asset ordinati sequenzialmente. Il primo di questi asset è di tipo receptionist: suo compito è quello di ricevere i messaggi che arrivano in input all'asset line, e di trasmetterli all'asset successivo. Ogni asset componente un asset di tipo line può invocare solo l'asset successivo (v. figura B.15).

- **Department**

  Un asset di tipo department è costituito da un certo numero di asset (che possono anche essere di tipo diverso), ognuno dei quali svolge una specifica funzione. Tra gli asset componenti un asset department deve esistere un asset di tipo receptionist, che di volta in volta riceve dall'esterno un messaggio, e lo inoltra ad un altro opportuno asset. L'eventuale messaggio di ritorno elaborato da questo asset viene inviato al receptionist, che poi lo inoltra all'esterno.

- **Division** e **Representative**

  L'asset di tipo division viene utilizzato per parallelizzare una procedura ricorsiva, progettata secondo il modello *divide-et-impera*.

  Un asset di questo tipo contiene un asset receptionist, e un asset representative, che rappresenta l'insieme delle chiamate ricorsive alla procedura associata all'asset division. Replicando l'asset representative, è possibile ottenere un'esecuzione parallela di questa procedura. Se con m si indica il fattore di replicazione, si ha una situazione come quella rappresentata in figura B.16.

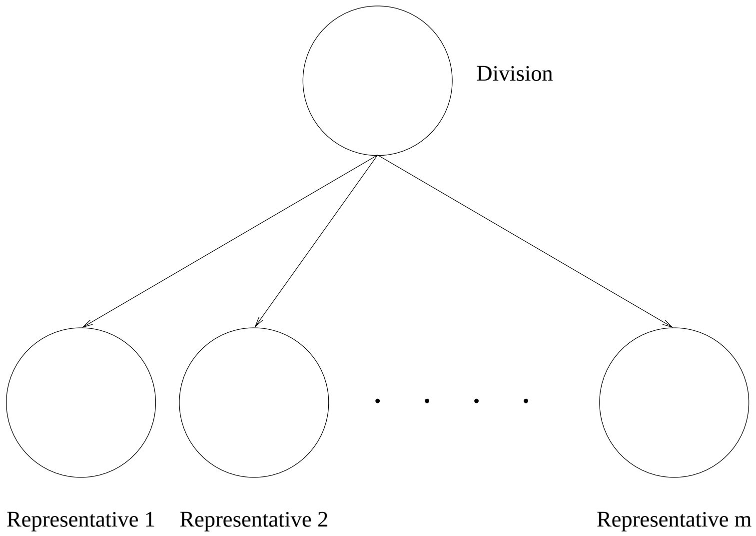


Figura B.16: Esempio di utilizzo degli asset division e representative

Per aumentare il grado di parallelismo, si può sostituire l'asset representative con un asset division, e replicare l'asset representative contenuto in quest'ultimo asset division, come rappresentato in figura B.17.

- **Receptionist**

  Come visto, la funzione svolta da un asset di tipo receptionist varia a seconda dell'asset composto in cui esso è inserito. In generale si può dire che l'asset receptionist è l'unico asset, tra quelli contenuti in quello composto a cui appartiene, ad essere "visibile" dall'esterno. Svolge infatti funzione di interfaccia tra gli asset contenuti nell'asset composto e tutti gli altri asset del programma.

  Un asset receptionist non può essere replicato.

- **Service**

  Ad un asset di tipo service è associata una procedura sequenziale che, a differenza di un asset di tipo individual, non può invocare nessun'altra procedura del programma. Inoltre essa può essere chiamata da una qualsiasi altra procedura del programma, indipendentemente dalle regole di visibilità degli asset del programma: questo può permettere la condivisione di variabili tra tutte le procedure del programma.

  Un asset service non può contenere altri asset.

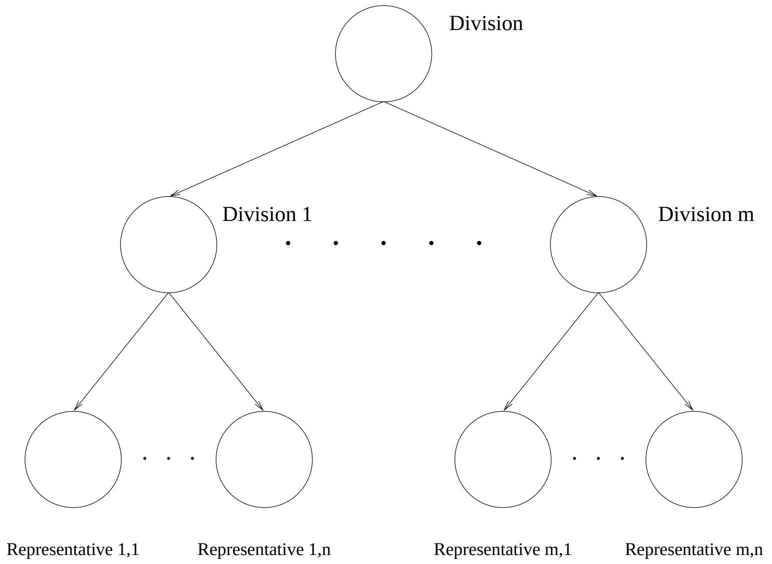


Figura B.17: Esempio di sostituzione di un asset division con un asset representative, per aumentare il grado di parallelismo

B.6.2 Compilazione

Una volta che la definizione dell'applicazione parallela è stata completata, attraverso la specifica dell'asset graph, e le varie procedure sono state codificate, il sistema integra il codice utente con chiamate a routine della libreria utilizzata (PVM, ISIS, o NMP), in modo da gestire le comunicazioni e le sincronizzazioni tra i vari processi. Il tutto viene poi compilato su ogni calcolatore costituente la macchina virtuale considerata.

B.6.3 Esecuzione

Quando l'applicazione parallela deve essere eseguita, viene creato un processo per ogni asset componente l'asset graph. Questi processi devono essere assegnati ai vari processori per la loro elaborazione: Enterprise individua quali calcolatori sono disponibili e quali invece stanno processando un job, in modo da bilanciare il load complessivo.

B.6.4 Tracing e debugging

Enterprise permette di "animare" l'esecuzione di un programma, in modo da verificare quali calcolatori vengono usati, qual è lo stato dei vari processi (idle, busy, wait), quanti e quali messaggi vengono scambiati tra i vari moduli, ecc...

L'animazione dell'esecuzione del programma è un utile strumento anche per il debugging: si possono infatti mettere in evidenza eventuali errori presenti nel codice, o nella struttura dell'applicazione.

Un altro strumento molto utile è la possibilità di registrare in un *event file* tutti gli eventi significativi avvenuti durante l'esecuzione. Utilizzando l'event file è poi possibile simulare una riesecuzione del programma, definendo dei breakpoint dove interrompere l'esecuzione, per poter esaminare il valore delle varie variabili.

B.7 Paralex

Paralex [8] [9] [12] [22] è un ambiente visuale per:

- la definizione
- la codifica
- l'esecuzione
- il tracing

di applicazioni parallele che devono essere eseguite in ambienti eterogenei e distribuiti, costituiti da una rete di workstation Unix, che usano ISIS [15] come sistema di message-passing. Paralex fornisce inoltre funzionalità per la gestione del fault tolerance e del load balancing dinamico.

B.7.1 Definizione della macchina virtuale

In Paralex prima di tutto è necessario definire la macchina virtuale che poi eseguirà l'applicazione parallela. Ogni calcolatore viene descritto attraverso un insieme di attributi, scelti dal programmatore; questi possono essere, ad esempio, il nome dell'architettura, il sistema operativo, un parametro che descrive le performance, ecc...

Una definizione della macchina virtuale può essere del tipo:

```
osde00      osf, perf=100, color, memory=128
osde02      osf, perf=90, memory=64
hpcms01     hpux, perf=100, color, memory=128
lnxpc01     linux, perf=50, memory=128
cmssun1     solaris, perf=110, memory=256
```

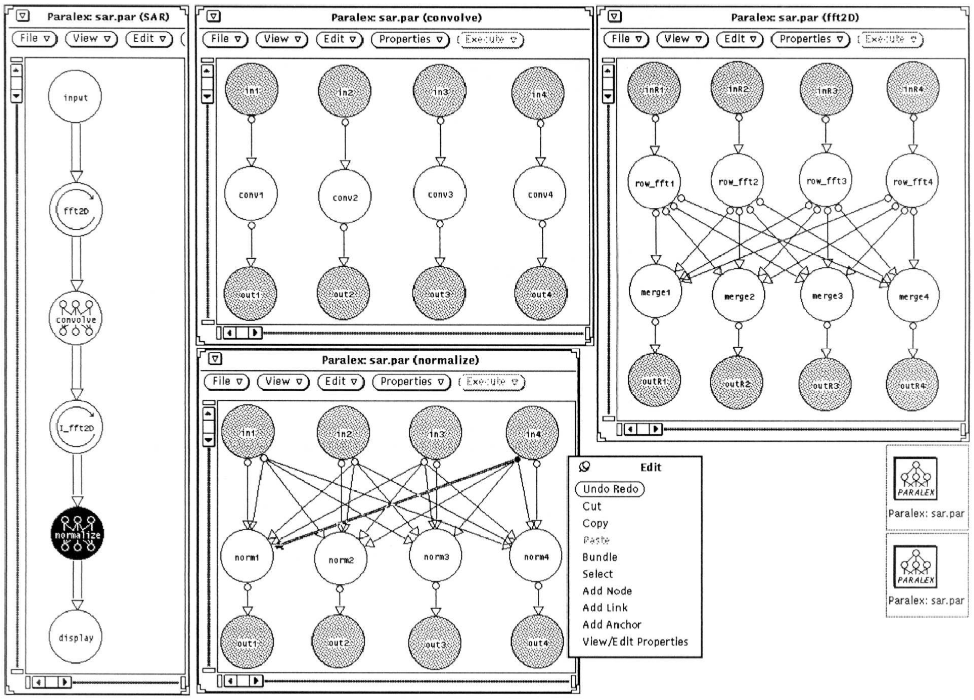


Figura B.18: Tool Paralex per la definizione di un'applicazione

Questi attributi vengono poi utilizzati dal sistema per la scelta dei calcolatori più adatti ad eseguire i vari moduli dell'applicazione parallela.

B.7.2 Definizione di un'applicazione

Attraverso un apposito tool grafico, mostrato in figura B.18, il programmatore definisce l'applicazione parallela attraverso un grafo diretto aciclico, in cui i nodi rappresentano unità computazionali, e gli archi definiscono "canali" attraverso i quali "passano" i dati tra i nodi connessi. La computazione associata ad un nodo può essere eseguita solo quando tutti gli archi incidenti quel nodo "contengono" valori.

Ogni nodo rappresenta una funzione, con uno o più input, e un solo output. Questa funzione deve essere "pura": non sono ammessi effetti collaterali, e non è possibile l'interazione con oggetti esterni, quali file, device, ecc...

Il codice associato a ciascun nodo può essere scritto in un qualsiasi linguaggio di programmazione, "compatibile" con il linguaggio C (cioè una procedura scritta in questo linguaggio deve poter essere invocata da un programma scritto in linguaggio C); è anche possibile associare ad un nodo un file binario eseguibile.

Per ogni nodo si devono specificare le caratteristiche del calcolatore che dovrà eseguirne il codice associato: questo viene fatto definendo un'espressione booleana, in cui si considerano gli attributi utilizzati nella definizione dei computer costituenti la macchina virtuale. In questo modo si può scegliere, ad esempio, il tipo di architettura del sistema che dovrà eseguire il modulo, le performance minime richieste per tale computer, ecc....

B.7.3 Compilazione

Una volta che il grafo di computazione è stato definito, ed è stato specificato il codice associato ai vari nodi, il programma può essere compilato.

Poiché si considera un ambiente eterogeneo, in cui i vari calcolatori possono usare diverse rappresentazioni interne dei dati, c'è la necessità di una "infrastuttura" comune, che consenta di rappresentare i dati allo stesso modo: a tal fine in Paralex viene utilizzato il toolkit ISIS.

L'applicazione parallela viene compilata remotamente sui diversi calcolatori che poi dovranno eseguirla.

B.7.4 Fault tolerance

Quando il programmatore definisce il grafo di computazione, ha la possibilità di associare ad esso un livello di *fault tolerance*: se viene specificato un livello di fault tolerance pari a *k*, Paralex genera l'opportuno codice in modo che ogni nodo del grafo venga replicato su *k+1* nodi distinti. Quindi, se vi sono fino a *k* fallimenti, il successo della computazione è comunque garantito.

Anche per la gestione del fault tolerance viene usato il toolkit ISIS.

B.7.5 Esecuzione

Prima che un programma possa essere eseguito, ogni nodo del grafo di computazione (ed eventualmente anche le sue repliche, se viene utilizzata la funzionalità di fault tolerance) deve essere associato ad un host del sistema distribuito considerato.

In questo problema di mapping, le unità elementari considerate non sono i singoli nodi, ma *catene*: una catena è definita come una sequenza di nodi che devono essere eseguiti sequenzialmente. Questi nodi devono essere "compatibili" tra loro, ovvero deve essere possibile eseguire il codice ad essi associato su uno stesso calcolatore.

Associare un'intera catena ad un nodo permette di ridurre l'overhead di comunicazione: in questo modo, infatti, il "traffico" tra i nodi che costituiscono la catena viene mantenuto locale.

L'algoritmo di mapping utilizzato da Paralex è di tipo euristico, in cui si considera un load balancing di tipo dinamico: periodicamente viene controllato il load di ogni macchina del sistema distribuito (anche il load non dovuto a job Paralex) e, se necessario, viene rivisto il mapping tra nodi e processori inizialmente adottato.

B.7.6 Tracing

Paralex permette all'utente di visualizzare lo stato di esecuzione di un'applicazione. È possibile controllare lo stato dei vari nodi, il nome dei calcolatori in cui essi sono in esecuzione, il flusso dei dati attraverso gli archi del grafo di computazione, ecc...

B.8 Meander

Meander [59] [60] è un insieme di tool integrati che permettono di gestire le varie fasi dello sviluppo di un'applicazione parallela:

- definizione del programma
- codifica
- compilazione
- esecuzione
- debugging

Meander può essere utilizzato per un sistema *transputer*, o per un ambiente costituito da un insieme omogeneo di workstation.

In figura B.19 è mostrato uno schema logico che descrive l'architettura del sistema Meander.

B.8.1 Definizione di un'applicazione

Un'applicazione parallela viene definita attraverso il cosiddetto *specification graph*: un grafo diretto costituito da *graph fragment*, connessi tra loro attraverso archi.

Un graph fragment è un sottografo, inserito nel grafo attraverso al più un arco entrante e un arco uscente di tipo *causal*. Gli archi di tipo causal definiscono il flusso sequenziale dei dati e delle informazioni di controllo.

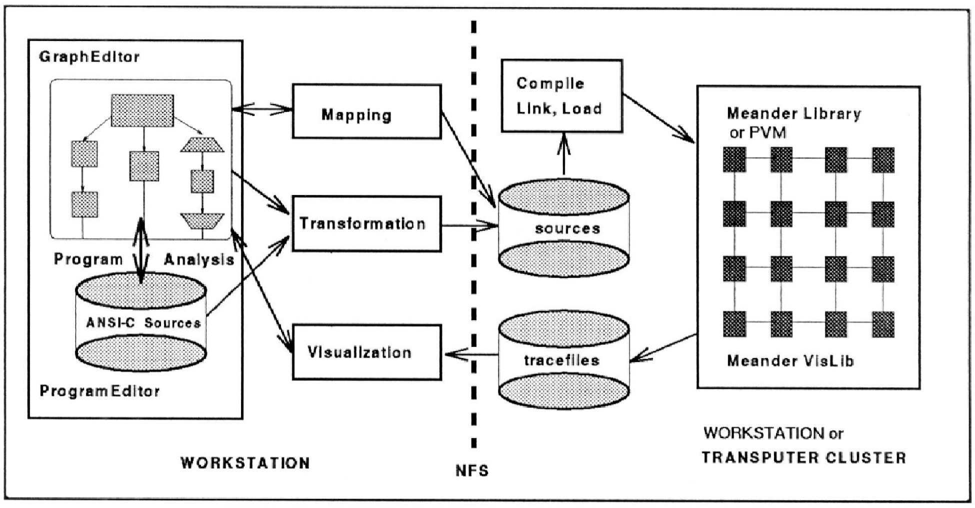


Figura B.19: Overview del sistema Meander

Il parallelismo dell'applicazione viene definito attraverso i nodi *create-child* e *wait-child*. Tra questi due nodi vengono inseriti, attraverso archi di tipo causal, una serie di graph fragment, che devono essere eseguiti in parallelo.

Ogni nodo dello specification graph rappresenta un processo sequenziale: ad esso viene associato una porzione di codice, scritta in linguaggio C.

Attraverso Meander è poi possibile definire altri costrutti, quali il costrutto di comunicazione, rappresentato attraverso i nodi *send* e *receive*, l'operazione condizionale, attraverso il costrutto *alt*, e l'iterazione, rappresentata dal costrutto *do*.

In figura B.20 è mostrato l'esempio di uno specification graph, realizzato attraverso l'apposito tool grafico di Meander.

B.8.2 Compilazione

Una volta che lo specification graph è stato definito, il codice scritto dal programmatore viene "integrato" con le chiamate alle apposite procedure che si occupano della gestione dei processi e della comunicazione. Nel caso di un ambiente costituito da una rete di workstation viene utilizzata la libreria PVM, mentre nel caso di sistemi transputer viene utilizzata un'altra specifica libreria.

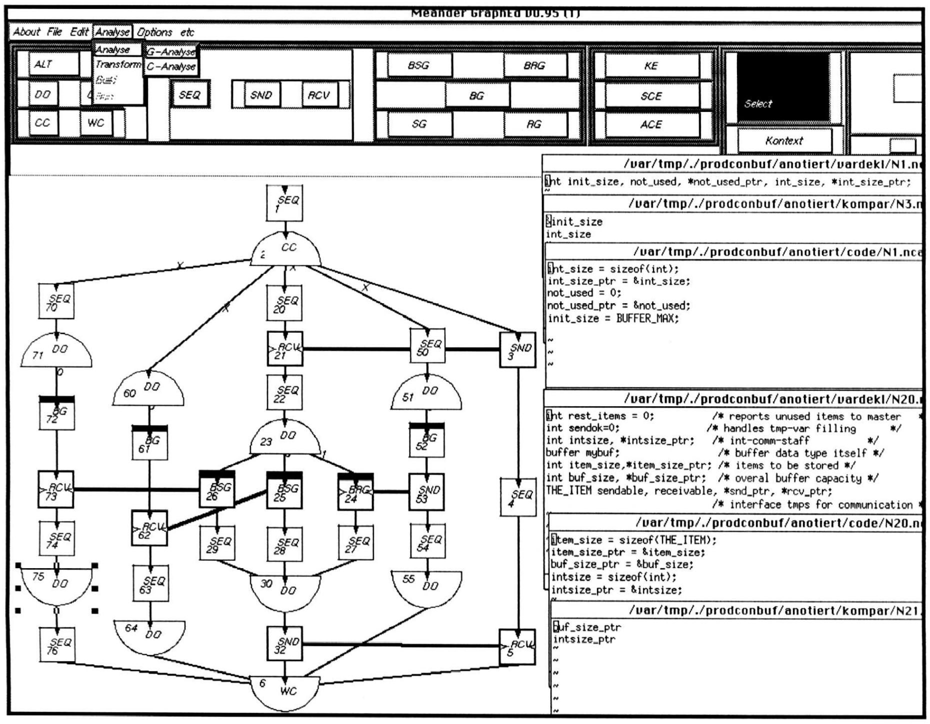


Figura B.20: Esempio di specification graph

B.8.3 Esecuzione

Per eseguire il programma parallelo è necessario associare i vari processi ai processori disponibili.

Per far questo, prima di tutto lo specification graph viene trasformato in un altro grafo, chiamato *process graph*: ogni nodo di questo grafo rappresenta un processo, mentre gli archi definiscono le comunicazioni tra questi processi. I vari nodi del process graph possono essere "annotati" con ulteriori parametri, che descrivono i requisiti necessari per l'esecuzione del codice associato (memoria richiesta, la necessità di particolari configurazioni hardware, ecc..)

Viene poi costruito un altro grafo, chiamato *hardware graph*, i cui nodi rappresentano i processori del sistema considerato, mentre gli archi definiscono le connessioni di rete che esistono tra essi. L'hardware graph può essere "annotato" dal programmatore con vari parametri, che descrivono le caratteristiche delle varie componenti (prestazioni dei processori, memoria presente, interfacce di rete usate, l'indicazione se i vari computer possono essere utilizzati in "presenza" di altri utenti, ecc...).

In figura B.21 viene mostrato un esempio di definizione di questi attributi per due calcolatori.

Le informazioni associate al process graph e all'hardware graph vengono utilizzate dal sistema, che automaticamente assegna ogni processo al processore più adatto.

B.8.4 Tracing e debugging

Meander permette di visualizzare l'esecuzione del programma in modalità off-line: quando il programma viene eseguito, viene prodotto un *tracefile*, dove vengono memorizzate le informazioni più significative. Il tracefile può essere poi utilizzato per simulare la ri-esecuzione del programma, che consente di esaminare lo stato dei vari nodi, il numero e la dimensione dei messaggi, il tempo impiegato da ogni processo per la computazione e per l'attesa di dati, ecc...

In figura B.22 è mostrato un esempio di visualizzazione dell'esecuzione di un programma.

Il tracing può essere uno strumento molto utile anche per il debugging e per il tuning dell'applicazione.

B.9 Considerazioni sugli ambienti esaminati

In questo paragrafo vengono valutate le caratteristiche dei vari ambienti considerati, in relazione con i requisiti richiesti per il sistema da realizzare.

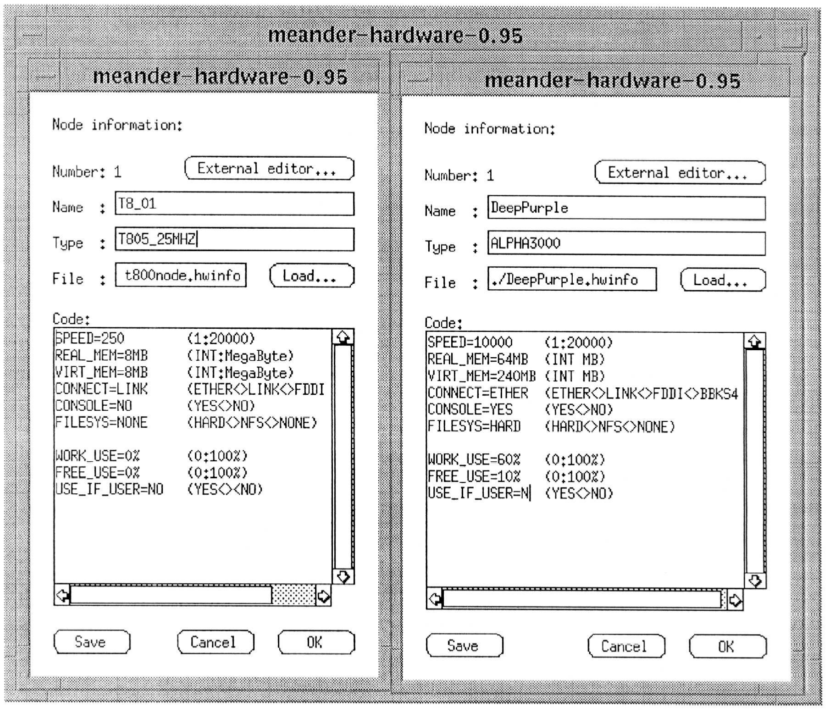


Figura B.21: Esempio di attributi di due nodi dell'hardware graph

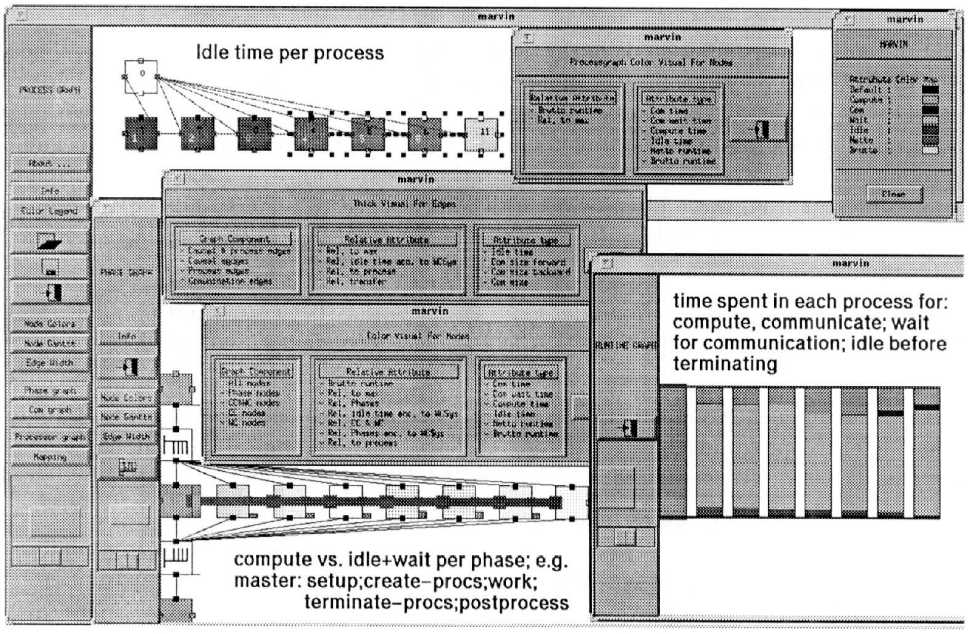


Figura B.22: Esempio di visualizzazione dell'esecuzione di un programma con Meander

L'analisi viene fatta esaminando singolarmente le diverse fasi della gestione dell'applicazione parallela, valutando, per ognuna di esse, le funzionalità disponibili nei diversi sistemi.

B.9.1 Ambiente di esecuzione

Come detto nel cap. 3, l'ambiente da considerare è una Wide Area Network, costituita da diversi siti connessi tra loro. Ogni sito rappresenta un metasistema, costituito da calcolatori collegati tra loro attraverso una rete locale di comunicazione.

Tutti gli ambienti esaminati, invece, fanno riferimento ad un ambiente di tipo LAN: non è contemplato il caso in cui i vari calcolatori sono posti in un ambiente geograficamente distribuito.

Per quel che riguarda le architetture dei calcolatori, è necessario considerare un ambiente eterogeneo costituito da workstation Unix e PC Windows NT.

La maggior parte degli ambienti analizzati (HeNCE, VPE, Enterprise, Paralex) possono essere utilizzati in un ambiente eterogeno di workstation Unix. Meander, invece, considera come ambiente di esecuzione un insieme omogeneo di calcolatori Unix. Infine in CODE l'applicazione può essere eseguita su una singola architettura MIMD. Allo stato attuale non esiste nessuna implementazione degli ambienti considerati per la piattaforma Windows NT.

B.9.2 Definizione dell'applicazione

Per quel che riguarda la definizione dell'applicazione, nel sistema da considerare deve essere possibile specificare le componenti sequenziali, e i vincoli di precedenza data-driven che esistono tra esse (un processo può iniziare l'esecuzione quando tutti i relativi input sono disponibili).

Questo è possibile in tutti gli ambienti esaminati, anche se con modalità diverse. L'unica limitazione è presente in HeNCE, in cui non si possono definire relazioni di precedenza tra nodi discendenti da nodi fratelli.

Nel sistema da realizzare i vari moduli dell'applicazione sono indipendenti tra loro. Non c'è necessità di interazione durante la loro esecuzione, e non è necessario prevedere l'uso di variabili globali, condivise tra le varie componenti sequenziali. Ogni modulo può avere zero o più input, e uno o più output. Ogni modulo è rappresentato da un programma binario eseguibile, o da uno script che definisce alcune variabili logiche, e che poi invoca un eseguibile.

Negli ambienti considerati, invece, ogni modulo sequenziale deve essere rappresentato attraverso una procedura scritta in linguaggio C (CODE, Enterprise, Meander), o a scelta in C o Fortran (HeNCE, VPE). Paralex invece consente di rappresentare ciascun modulo attraverso una procedura scritta in un linguaggio "compatibile" con il linguaggio C, o attraverso un programma binario eseguibile. In Paralex, però, ciascun modulo deve rappresentare una funzione con un solo output; inoltre tale funzione deve essere "pura", ossia essa non può manipolare nessun oggetto esterno (file, device, ecc...), cosa che invece deve essere possibile nel sistema DEMI da realizzare.

In nessuno degli ambienti esaminati viene considerato il problema della gestione dei dati: la loro archiviazione, il loro tipo, l'indicazione se i risultati intermedi devono essere salvati o se devono essere cancellati, ecc... In DEMI, invece, la gestione dei dati è un aspetto molto importante del problema.

B.9.3 Esecuzione del programma

Una volta che l'applicazione è stata definita e compilata, i vari moduli devono essere assegnati ai vari calcolatori per la loro esecuzione. L'obiettivo primario di questo problema di mapping è la massimizzazione del throughput.

I vari ambienti esaminati gestiscono questo problema di mapping in diversi modi: in alcuni sistemi (ad esempio VPE) il programmatore deve specificare il calcolatore o il tipo di calcolatore che deve eseguire il modulo (nel secondo caso per l'esecuzione viene scelto semplicemente il primo computer "libero" di quel tipo), in altri (come HeNCE) il programmatore deve specificare il "costo" dell'esecuzione dei vari moduli sui vari calcolatori (in questo caso il mapping viene effettuato in modo da minimizzare il costo totale di esecuzione). Paralex usa un algoritmo più sofisticato di load balancing dinamico, in cui, a differenza degli altri ambienti, si considera anche il load dovuto a job di utenti generici. Paralex, inoltre, consente di minimizzare l'overhead di comunicazione, associando ai vari processori catene di task, anziché singoli task.

Nel sistema DEMI, in presenza di utenti generici, in generale l'esecuzione dei moduli dell'applicazione parallela deve essere sospesa: gli utenti devono avere più alta priorità nell'utilizzo delle risorse di calcolo. Tra gli ambienti considerati solo Meander considera questo aspetto.

Altre funzionalità aggiuntive implementate in alcuni sistemi (come la gestione del fault tolerance prevista in Paralex) non sono richieste per il sistema da realizzare.

In nessuno degli ambienti esaminati sono implementate funzionalità quali il *checkpointing* e il *process migration*, che possono essere utili strumenti per realizzare la massimizzazione del throughput. Utilizzando il meccanismo del checkpointing, infatti, si ha la garanzia che un job viene sicuramente completato: anche se, ad esempio, vi è il crash di un sistema che sta eseguendo un job, l'elaborazione eseguita non viene comunque "persa". Il meccanismo di fault tolerance implementato nel sistema Paralex (in cui l'esecuzione del job viene replicata su più processori), invece, non consente di massimizzare il throughput (anzi, in generale, usando questa tecnica il throughput diminuisce).

La migrazione dei processi tra diversi computer della stessa architettura (in cui si utilizza il meccanismo del checkpointing), è un utile strumento per gestire i casi in cui utenti generici vogliono avere un utilizzo esclusivo delle loro workstation: quando un utente generico "reclama" l'uso di una workstation, i job in esecuzione su questo calcolatore possono essere migrati su un altro disponibile.

B.9.4 Monitor dell'esecuzione

Quasi tutti gli ambienti considerati (tranne CODE e VPE) prevedono una funzionalità che consente al programmatore di visualizzare lo stato di esecuzione del programma. Alcuni di essi (come Enterprise e Meander), consentono anche un'analisi off-line, per cui si può simulare la ri-esecuzione del programma, definendo dei breakpoint per controllare il valore delle variabili.

Nel sistema da realizzare è richiesto un supporto molto limitato per questo aspetto: è sufficiente poter verificare su che computer i vari moduli sono in esecuzione.

B.9.5 Conclusioni

Non esiste, quindi, un ambiente che presenti tutti i requisiti e tutte le funzionalità richieste. In alcuni sistemi sono implementate certe funzionalità, ma non ne sono presenti altre, indispensabili per il sistema da realizzare.

Altre funzionalità (ad esempio il supporto per l'architettura Windows Nt, la possibilità di gestire i dati, ecc...), invece, non sono implementate in nessuno degli ambienti esaminati.

Appendice C

Il modello dei dati a oggetti

I concetti fondamentali da definire per il modello dei dati a oggetti sono quelli di oggetto, di tipo di oggetto, di classe, di gerarchie fra tipi e gerarchie tra classi.

C.1 Oggetti

Gli oggetti vengono utilizzati per rappresentare le entità dell'universo del discorso. Più precisamente per *oggetto* si intende un'entità software con un proprio stato (rappresentato da un insieme di variabili e/o costanti, chiamati *campi* o *componenti*) e con una propria identità, immutabile, che rende un oggetto diverso da ogni altro oggetto. L'identità di un oggetto non è legata al valore del suo stato.

Ogni oggetto ha anche un "comportamento", rappresentato da un insieme di procedure locali, dette *metodi*, che modellano le operazioni che lo riguardano. Le operazioni applicabili ad un oggetto si dicono *messaggi*: un oggetto "risponde" ad un messaggio utilizzando i propri metodi, ed eventualmente modificando il proprio stato. L'insieme dei messaggi a cui un oggetto può rispondere, e l'insieme dei campi "accessibili" dall'esterno definiscono la sua *interfaccia*.

Ogni oggetto ha un certo tipo: un tipo oggetto definisce i suoi possibili valori e gli operatori ad essi applicabili.

I nomi dei messaggi e i nomi dei campi accessibili dello stato vengono chiamati *attributi* dell'oggetto. Un attributo può essere:

- *atomico*, se il suo valore non è scomponibile (es. il nome di una persona), o altrimenti *strutturato* (es. l'indirizzo, scomponibile in via, numero civico, ecc...);

- *unione*, se può assumere valori di tipo diverso (ad esempio, l'attributo 'mezzo

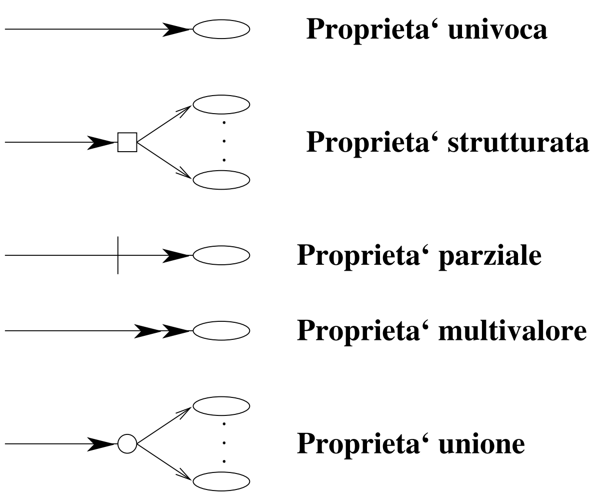


Figura C.1: Rappresentazione di tipi di attributi

utilizzato' di un tipo oggetto 'viaggio' può essere di tipo 'aereo', di tipo 'treno', ecc...), o in caso contrario *semplice*;

- *univoco*, se il suo valore è unico (es. il cognome di una persona), o altrimenti *multivalore* (ad esempio le carte di credito di una persona: una persona può avere più carte di credito);
- *totale* (es. il cognome di una persona), o altrimenti *parziale* (ad esempio l'attributo 'carta di credito' di un tipo 'persona' è un attributo parziale, in quanto una persona può non avere carta di credito);
- *costante*, se il suo valore non cambia nel tempo (es. la data di nascita), o altrimenti *modificabile*.

Nel formalismo grafico adottato, un tipo oggetto viene rappresentato attraverso un'ellisse. In figura C.1 sono rappresentate le simbologie che si utilizzano per rappresentare alcuni tipi di attributi.

C.2 Classi

Gli oggetti rappresentano le entità, mentre le *classi* modellano collezioni di entità: una classe rappresenta un insieme di oggetti dello stesso tipo, modificabile con operatori per l'inclusione e operatori per la rimozione di elementi dall'insieme.

Una classe può essere rappresentata graficamente attraverso un rettangolo, eventualmente indicando gli attributi dei suoi elementi, come indicato nell'esempio

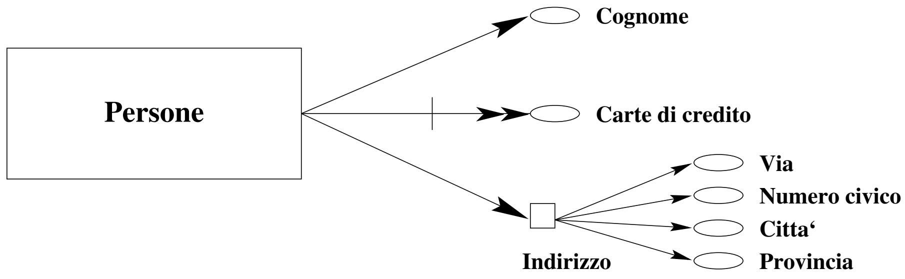


Figura C.2: Esempio di rappresentazione grafica di una classe

di figura C.2 (l'attributo 'Cognome' è un attributo univoco, 'Carte di credito' è un attributo multivalore e parziale, 'Indirizzo' è un attributo strutturato).

C.3 Gerarchie

Dati due tipi oggetti T_1 e T_2, si dice che T_1 è sottotipo di T_2 (o T_2 è supertipo di T_1) quando gli oggetti di tipo T_1 possono essere usati in qualsiasi contesto in cui è previsto un oggetto di tipo T_2.

Ad esempio i tipi oggetti *Impiegato* e *Studente* possono essere considerati sottotipi del tipo oggetto *Persona*: un oggetto di tipo *Impiegato* o di tipo *Studente* può infatti essere visto anche come un oggetto di tipo *Persona*.

Se il tipo oggetto T_1 è sottotipo del tipo oggetto T_2, gli oggetti di tipo T_1 ereditano tutti gli attributi del corrispondente supertipo T_2. Nel sottotipo è comunque possibile definire nuovi attributi, e/o ridefinire attributi ereditati dal supertipo (si parla in questo caso di *overriding*).

Un sottotipo può essere definito a partire da un unico supertipo (*ereditarietà semplice*) o da più supertipi (*ereditarietà multipla*); ad esempio il tipo oggetto *Impiegata* è sottotipo sia del tipo oggetto *Lavoratore*, che del tipo oggetto *Donna*.

È possibile definire una gerarchia anche tra le classi. Date due classi C_1 e C_2, se C_1 è sottoclasse di C_2 (C_2 è una superclasse di C_1) allora:

- gli elementi di C_1 sono un sottoinsieme degli elementi di C_2 (*vincolo estensionale*);
- il tipo degli oggetti di C_1 è sottotipo del tipo degli oggetti di C_2 (*vincolo strutturale*).

Quindi, visto il vincolo estensionale, se si aggiunge un oggetto ad una sottoclasse, questo elemento fa automaticamente parte anche della corrispondente superclasse.

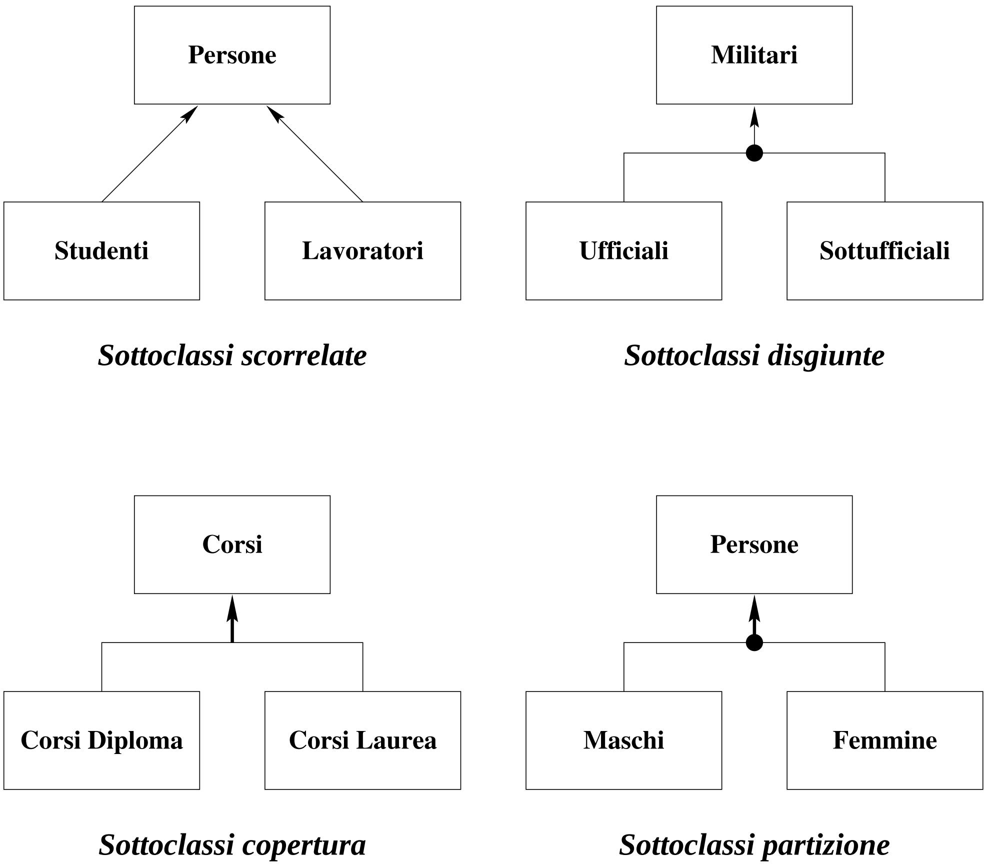


Figura C.3: Definizioni di sottoclassi

Un insieme di sottoclassi di una stessa classe soddisfano il *vincolo di disgiunzione* se ogni coppia di sottoclassi dell'insieme è disgiunta (*sottoclassi disgiunte*). Un insieme di sottoclassi soddisfa il *vincolo di compertura* se l'unione degli elementi delle sottoclassi coincide con l'insieme degli elementi della superclasse (*sottoclassi copertura*). Se entrambi i vincoli sono soddisfatti, si parla di *sottoclassi partizione* della superclasse. Se invece tutti e due i vincoli non sono soddisfatti, si parla di *sottoclassi scorrelate.*

In figura C.3 vengono mostrate le rappresentazioni grafiche che si utilizzano per questi tipi di sottoclassi (quindi il vincolo di disgiunzione viene rappresentato con un pallino nero, mentre il vincolo di copertura con una freccia grossa).

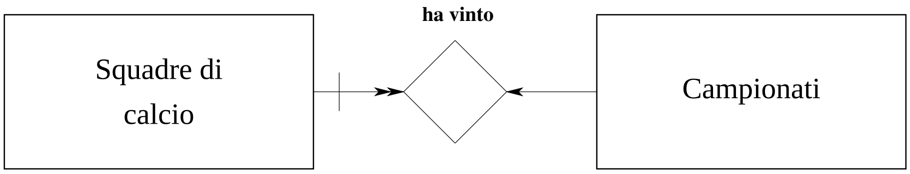


Figura C.4: Esempio di rappresentazione di un'associazione

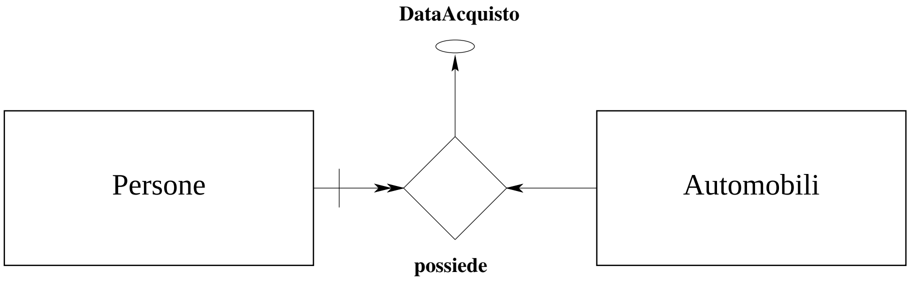


Figura C.5: Esempio di associazione con attributi

C.4 Associazioni

Nel formalismo grafico considerato, un'associazione tra classi viene rappresentata con un rombo collegato con archi alle classi considerate. Il rombo viene "etichettato" con il nome di un predicato, considerando una frase del tipo "soggetto-predicato-complemento", ottenuta leggendo il diagramma da sinistra a destra (o dall'alto in basso): il soggetto è un generico elemento della prima classe, e il complemento è un generico elemento della seconda classe. Ad esempio, con riferimento alla figura C.4, il predicato 'ha vinto' fa riferimento alla frase 'una squadra di calcio ha vinto un campionato'.

Un'associazione (diretta o inversa) univoca viene rappresentata con una freccia singola, mentre un'associazione multivalore viene definita attraverso una freccia doppia. Un'associazione parziale si rappresenta con un "taglio" nel corrispondente arco: se non c'è il taglio, l'associazione è totale.

Nell'esempio di figura C.4, è rappresentata l'associazione tra le classi *Squadre di calcio* e *Campionati*: la relazione diretta è parziale (una squadra potrebbe non aver mai vinto un campionato) e multivalore (una squadra di calcio potrebbe aver vinto più campionati); la relazione inversa è univoca e totale (un campionato è vinto da una ed una sola squadra).

Se l'associazione ha delle proprietà, al rombo si aggiungono archi etichettati con i corrispondenti nodi, come rappresentato nell'esempio di figura C.5.

Appendice D

L'API ODBC

L'API ODBC (*Open Data Base Connectivity*) [6] [37] è attualmente il più diffuso standard per l'utilizzo di DBMS relazionali: è stato definito da Microsoft nei primi anni "90, considerando lo standard SQL CLI (*Call Level Interface*), proposto dai comitati X/Open e ISO/IEC.

Attraverso l'API ODBC è possibile operare su un database relazionale in modo "uniforme", indipendentemente da quali siano il DBMS e il sistema operativo considerati.

Un'applicazione che usa questa API può accedere a diversi tipi di DBMS (anche simultaneamente), senza necessità di modificare il codice del programma, e senza bisogno di una ricompilazione o di un relink.

L'architettura ODBC è mostrata in figura D.1.
Vi sono quindi quattro componenti fondamentali:

- l'applicazione;
- il Driver Manager;
- i driver;
- i DBMS.

L'applicazione è il programma che deve accedere al DBMS per operazioni di inserimento dati, modifica, cancellazione e ricerca. Queste operazioni vengono realizzate attraverso chiamate a funzioni ODBC, in cui si invoca l'esecuzione di comandi SQL.

L'applicazione invoca le funzioni ODBC di un particolare modulo, chiamato *Driver Manager*: una libreria che gestisce la comunicazione tra l'applicazione e i driver: ad ogni tipo di DBMS è associato uno specifico driver. Quando un'applicazione chiama una funzione ODBC del Driver Manager, quest'ultimo invoca la corrispondente routine del driver specifico al DBMS considerato.

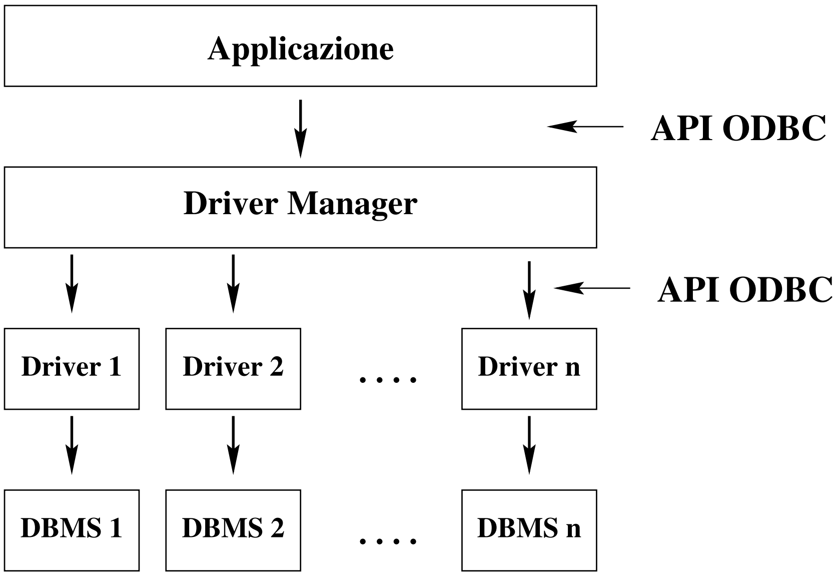


Figura D.1: Architettura ODBC

I driver sono librerie in cui sono implementate le funzioni ODBC: attraverso queste routine si gestisce la connessione con il DBMS, si eseguono i comandi SQL, si esegue un controllo sugli errori, si convertono i dati nel formato richiesto dall'applicazione, ecc...
L'API ODBC gestisce quindi la comunicazione sia tra l'applicazione e il Driver Manager, che tra il Driver Manager e ogni driver.

Il driver e il DBMS possono risiedere sullo stesso calcolatore, o su computer diversi. Ad esempio nella configurazione di figura D.2, sul computer *Host A* è presente l'applicazione, il Driver Manager e il driver di uno specifico DBMS, che invece è installato sul computer *Host B*.

Figura D.2: Esempio di configurazione

Attraverso l'API ODBC è sempre possibile utilizzare il sottoinsieme del linguaggio SQL implementato in tutti i sistemi relazionali. Possono comunque esistere estensioni non standard presenti su di uno specifico sistema: vi sono apposite funzioni ODBC che permettono di determinare quali funzionalità sono implementate in un DBMS, per poi utilizzarle.

Bibliografia

[1] *http://delphisrv01.pd.infn.it/demi.*

[2] *http://users.ids.net/ bjepson/freeODBC.*

[3] *http://wwwcn.cern.ch/ offline/physics/pubdet2.html.*

[4] *http://wwwcn.cern.ch/pdp/benchmark.*

[5] *http://www.tcx.se.*

[6] A. Albano, G. Ghelli, e R. Orsini, *Basi di Dati Relazionali e a Oggetti*, Zanichelli, 1997.

[7] J. N. Albert, J. E. Augustin, Y. Belokopytov, F. Carena, F. Costa, R. Gokieli, J. C. Marin, M. Mazzucato, M. Michelotto, L. Pape, V. Perevozchikpv, M. E. Pol, e R. Rampazzo, *DELFARM 91 - Part 1: Hardware and Software Overview*, May 1993.

[8] L. Alvisi, A. Amoroso, Ö. Babaoğlu, A. Baronio, R. Davoli, e L. Giachini, *Parallel Scientific Computing in Distributed Systems: The Paralex Approach*, 1991.

[9] L. Alvisi, Ö. Babaoğlu, R. Davoli, e A. Amoroso, *Mapping Parallel Computations on to Distributed Systems in Paralex*, Technical Report UBLCS-92-1, University of Bologna, Laboratory for Computer Science, January 1994.

[10] M. Athanas e M. Ogg, *An Evaluation of PCs for High Energy Physics under Windows NT and Linux*, February 1997.

[11] P. Avery, *Analysis Session Summary for CHEP95*, 1995.

[12] Ö. Babaoğlu, L. Alvisi, A. Amoroso, R. Davoli, e L. A. Giachini, *Run-time Support for Dynamic Load Balancing and Debugging in Paralex*, Technical Report TR91-1251, Cornell University, Computer Science Department, December 1991.

[13] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, K. Moore, P. Netwon, e V. Sunderam, *HeNCE: A Users' Guide Version 2.0*, 1994.

[14] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, e V. S. Sunderam, *Graphical Development Tools for Network-based Concurrent Supercomputing*, 1991.

[15] K. Birman, *The ISIS System Manual, Version 2.1, ISIS Project*, 1991.

[16] A. Bricker, M. Litzkow, e M. Livny, *Condor Technical Summary*, Technical Report 1069, Computer Sciences Department, University of Wisconsin-Madison, USA, January 1992.

[17] J. C. Browne, *Formulation and Programming of Parallel Computations: A Unified Approach*, Proceeding of the 1985 International Conference on Parallel Processing, (1985), pp. 624–631.

[18] J. C. Browne, J. J. Dongarra, S. I. Hyder, K. Moore, e P. Netwon, *Visual Programming and Parallel Computing*, Technical Report CS-94-229, University of Tennessee, April 1994.

[19] B. D. Burow, *Funnel: Towards Comfortable Event Processing*, (1995).

[20] G. Cassel, K. Birman, A. Ricciardi, M. Ogg, K. Marzullo, P. Avery, e J. Davis, *Distribuited Computing and Databases for High Energy Physics*, June 1993.

[21] CraySoft, Cray Research Inc., *Introducing NQE*, 1997.

[22] R. Davoli, L. A. Giachini, Ö. Babaoğlu, A. Amoroso, e L. Alvisi, *Parallel Computing in Networks of Workstations with Paralex*, IEEE Transactions on Parallel and Distributed Systems, 7 (1996), pp. 371–384.

[23] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, e J. Pruyne, *A Worldwide Flock of Condors: Load Sharing Among Workstation Clusters*, Technical Report DUT-TWI-95-130, Delft University of Technology, Department of Technical Mathematics and Informatics, Delft, The Netherlands, 1995.

[24] R. F. Freund e H. J. Siegel, *Heterogeneous Processing*, Computer, 26 (1993), pp. 13–17.

[25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, e V. S. Sunderam, *PVM 3 User's Guide and Reference Manual*, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, September 1994.

[26] Genias Software GmbH, *Codine, Computing in Distributed Networked Environments, Installation and Administration Guide 4.0*, Erzgebirgstr.2B, D-93073 Neutraubling, Germany, March 1997.

[27] A. Ghafoor e J. Yang, *A Distributed Heterogeneous Supercomputing Management System*, Computer, 26 (1993), pp. 78–86.

[28] C. Ghezzi, A. Fuggetta, S. Morasca, A. Morzenti, e M. Pezzè, *Ingegneria del Software: Progettazione, Sviluppo e Verifica*, Mondadori Informatica, Milano, 1991.

[29] IBM Corporation, *Using and Administering IBM LoadLeveler, Release 3.0*, August 1996.

[30] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, e C.-L. Wang, *Heterogeneous Computing: Challenges and Opportunities*, Computer, 26 (1993), pp. 18–27.

[31] E. Knudstrup, *Samba*, Sys Admin: The Journal for UNIX Systems Administrators, 5 (1996), pp. 8–14.

[32] M. Litzkow e M. Livny, *Experience with the Condor Distributed Batch System*, in Second IEEE Workshop on Experimental Distributed Systems, Huntsville, Alabama, October 1990, Institute of Electrical and Electronics Engineergs (IEEE), pp. 97–101.

[33] M. Litzkow, M. Livny, e M. Mutka, *Condor: A Hunter of Idle Workstations*, Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, (1988).

[34] M. Livny e M. Melman, *Load Balancing in Homogeneous Broadcast Distributed Systems*, in Proc. Modeling Perform. Eval. Comput. Syst., ACM, April 1982, pp. 47–55.

[35] T. Marsland, T. Breitkreutz, e S. Sutphen, *A Network Multiprocessor for Experiments in Parallelism*, Concurrency - Practice and Experience, 3 (1991), pp. 203–219.

[36] P. Mazzanti, M. Livny, F. Brasolin, e D. Bortolotti, *Condor: A High Throughput System. The INFN Bologna Condor Pool*, Technical Report IFUB96/22, INFN and Department of Physics, Bologna, Italy, 1996.

[37] Microsoft Corporation, *ODBC 3.0 Programmer's Reference*, 1997.

[38] B. Muangkhot, *Is DFS the Distributed File System of the 90s ?*, ConneXions, The Interoperability Report, 7 (1993), pp. 29–31.

[39] P. Netwon e J. C. Browne, *The CODE 2.0 Graphical Parallel Programming Language*, 1992.

[40] P. Newton e J. Dongarra, *Overview of VPE: A Visual Environment for Message-Passing, Parallel Programming*, Technical Report CS-94-261, University of Tennessee, November 1994.

[41] P. W. Newton, *A Graphical Retargetable Parallel Programming Environment and its Efficient Implementation*, Technical Report CS-TR-93-28, University of Texas, Austin, December 1993.

[42] Objectivity Inc., *Objectivity Technical Overview*, June 1997.

[43] Platform Computing Corporation, *LSF Batch Administrator's Guide*, North York, Ontario, Canada, December 1997.

[44] ——, *LSF Batch User's Guide*, North York, Ontario, Canada, December 1997.

[45] ——, *LSF Programmer's Guide*, North York, Ontario, Canada, December 1997.

[46] J. Pruyne e M. Livny, *Providing Resource Management Services to Parallel Applications*, Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing.

[47] ——, *Parallel Processing on Dynamic Resources with CARMI*, Lecture Notes in Computer Science, 949 (1995), pp. 259–266.

[48] A. Ricciardi, M. Ogg, e E. Rothfus, *The NILE System Architecture: Fault-Tolerant, Wide-Area Access to Computing and Data Resources.*

[49] F. J. Rinaldo e M. R. Fausey, *Event Reconstruction in High-Energy Physics*, Computer, 26 (1993), pp. 68–77.

[50] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, e B. Lyon, *Design and Implementation of the Sun Network File System*, Summer 1985.

[51] M. Satyanarayanan, *Scalable, Secure, and Highly Available Distributed File Access*, IEEE Computer, 23 (1990), pp. 9–21.

[52] J. Schaeffer, D. Szafron, G. Lobe, e I. Parsons, *The Enterprise Model for Developing Distributed Applications*, IEEE Parallel and Distributed Technology, 1 (1993), pp. 85–96.

[53] R. Schiefer e D. Francis, *Parallelisation of an Existing High Energy Physics Event Reconstruction Software Package*, IEEE Trans. Nucl. Sci., 43 (1996), p. 79.

[54] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, e J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge MA, USA, 1996.

[55] Supercomputer Computations Research Institute, Florida State University, *DQS 3.1.3 User Guide*, March 1996.

[56] The RD45 collaboration, *Status Report of the RD45 Project*, Technical Report CERN/LHCC 98-11, CERN, Geneva, Switzerland, April 1998.

[57] J. Wang, S. Zhou, K. Ahmed, e W. Long, *LSBATCH: A Distributed Load Sharing Batch System*, Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, April 1993.

[58] R. W. Watson e R. A. Coyne, *The Parallel I/O Architecture of the High-Performance Storage System (HPSS)*, in Proceedings of the Fourteenth IEEE Simposium on Mass Storage Systems, IEEE Computer Society Press, September 1995, pp. 27–44.

[59] G. Wirtz, *A Visual Approach for Developing, Understanding and Analysing Parallel Programs*, in Proceedings of the 1993 IEEE Symposium on Visual Languages, E. P. Glinert e K. A. Olsen, eds., Bergen, Norway, August 1993, IEEE Computer Society Press, pp. 261–266.

[60] ——, *A Visual Toolset for Message-Passing Parallel Programming*, in Proceedings of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '96), H. R. Arabnia, ed., vol. I, Sunnyvale, CA, August 1996, CSREA, pp. 301–312.

[61] S. Zhou, J. Wang, X. Zheng, e P. Delisle, *Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems*, Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto, April 1992.