Università degli Studi di Perugia
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

Studio di un prototipo per la gestione dinamica di ambienti virtuali.

Candidato
Riccardo Maria Cefalà

Relatore
Prof. **Leonello Servoli**

Correlatore
Mirko Mariotti

Anno Accademico
2006-2007

Indice

Introduzione

Questo elaborato presenta il lavoro di tesi e stage svolto presso la sede INFN di Perugia e il Dipartimento di Fisica dell'Università degli Studi di Perugia a partire dal Novembre 2007.

Sinossi

I moderni sistemi di calcolo distribuito, rappresentati in primis dai Cluster di Computer e dalle Griglie Computazionali di ancor più recente introduzione, definiscono oggi lo stato dell'arte nel calcolo ad alte prestazioni e non solo.

Accanto alle promesse di elevate capacità di calcolo e offerta di servizi, emergono tuttavia nuove problematiche un tempo non rilevanti per i sistemi più centralizzati e di minori dimensioni: già in cluster medio-grandi il problema dell'eterogeneità che sopraggiunge a livello sia hardware che software durante il ciclo di vita del sistema, raggiunge proporzioni notevoli e viene ulteriormente amplificato nelle realtà emergenti delle griglie computazionali; esse devono infatti essere in grado di aggregare un gran numero di risorse eterogenee, concertandone l'utilizzo per offrire molte tipologie di servizi.

Inoltre, le organizzazioni e le entità che compartecipano alla creazione di sistemi di calcolo condivisi, presentano spesso necessità molto stringenti e la convivenza, ovvero il soddisfacimento contestuale dei bisogni, risulta difficilmente praticabile, se non attraverso l'incremento del numero di macchine dedicate. Tuttavia, i tempi di inutilizzo dei servizi richiesti o lo scarso carico computazionale che eventualmente essi comportano, fanno sì che le risorse riservate alla loro offerta risultino inaccettabilmente sotto-utilizzate. Ne consegue un aumento degli oneri di gestione sia in termini di risorse umane che di costi.

La possibilità d'astrazione tramite virtualizzazione dalle macchine fisiche, consente la definizione di ambienti d'esecuzione virtuali migrando su di essi i servizi: una gestione dinamica di tali ambienti, che li attivi e distrugga tenendo in conto le necessità contingenti, insieme all'impiego delle efficenti e ormai consolidate tecniche di para-virtualizzazione, offrirebbe le premesse

per un coordinamento più flessibile e rispondente ai bisogni, diminuendo drasticamente il tasso di inutilizzo delle risorse.

Il prototipo oggetto di questa tesi mira alla realizzazione di un sistema che sia in grado di gestire più classi di ambienti virtuali, attraverso il monitoring delle risorse disponibili e sulla base di criteri di decisione rispondenti alle necessità.

Struttura della Tesi

La tesi è strutturata in cinque capitoli:

Capitolo Primo: Sono esposti i sistemi di calcolo distribuito, con particolare attenzione ai Cluster di Computer e con un'introduzione all'architettura e alle problematiche del Cluster INFN della sede di Perugia e delle Griglie Computazionali.

Capitolo Secondo: Si introducono i concetti chiave dei sistemi di virtualizzazione con riferimento alle basi teoriche, descrivendone le caratteristiche e i vantaggi. Vengono mostrate in particolare le caratteristiche del sistema di para-virtualizzazione Xen.

Capitolo Terzo: Vengono definite più chiaramente le problematiche affrontate dal prototipo, dando una prima descrizione dell'architettura dello stesso, introducendo il concetto di ambiente virtuale e dell'approccio seguito per l'implementazione.

Capitolo Quarto: È descritta la realizzazione delle componenti del prototipo, insieme a presentarne le scelte implementative alla base. Viene inoltre mostrato l'ambiente di test nel quale il prototipo è stato implementato.

Capitolo Quinto: Presenta le conclusioni finali insieme a mostrare alcuni degli sviluppi futuri del prototipo e alcune considerazioni personali.

Nell'**Appendice A** sono raccolte le specifiche seguite per la realizzazione dell'ambiente di test.

Nell'**Appendice B** è contenuto il codice sorgente del prototipo.

Capitolo 1

Cluster Computing

1.1 Introduzione

I progressi in molti dei campi di ricerca che forniranno le soluzioni agli odierni problemi saranno incontrovertibilmente determinati dalla capacità di sfruttare le possibilità fornite dal calcolo ad alte prestazioni.

Ciò è affermabile sulla base della considerazione secondo cui molti dei più attesi ed esaltanti risultati delle ricerche in campi quali la fisica delle alte energie, la scienza medica, l'esplorazione dello spazio, le ricerca sul clima, le scienze dei materiali, l'intelligenza artificiale e la gestione dell'informazione e delle comunicazioni, sono strettamente legati al progresso delle performance dei calcolatori.

Ma come è possibile ottenere prestazioni sempre maggiori? Negli scorsi decenni abbiamo assistito a enormi passi in avanti ottenuti attraverso:

- Miglioramento delle architetture e delle prestazioni dei processori.
- Algoritmi e tecniche di programmazione più efficienti.

Tuttavia ad uno sguardo più critico non possono sfuggire i limiti di questi due approcci: gli odierni processori al silicio raggiungeranno nel prossimo futuro i limiti fisici dettati da fattori dovuti all'integrazione sempre più spinta dei transistor e inoltre non si potranno tralasciare gli alti costi per la realizzazione di architetture specializzate; d'altra parte, anche la ricerca sugli algoritmi, per quanto di fondamentale importanza, non può esimere dalla necessità di cruda potenza di calcolo di molti problemi dei quali si occupano le discipline sopra citate.

Fortunatamente, oltre alle due strade indicate ne emerge una terza e cioè quella offerta dalla possibilità di distribuire il carico computazionale coordinando l'utilizzo di più unità di elaborazione per un singolo scopo.

Questo concetto può essere considerato come l'idea alla base del calcolo parallelo.

1.2 Sistemi di Calcolo: Tassonomia di Flynn

Il proponimento di parallelizzare i calcoli per la risoluzione dei problemi non è nuovo, si può anzi affermare che è piuttosto datato considerando la storia relativamente breve dei sistemi di calcolo.

È infatti del 1967 l'articolo nel quale Gene Amdahl introduce la legge che a tutt'oggi viene utilizzata per predire il massimo aumento prestazionale teorico ottenibile aumentando il numero di processori, gettando quelle che sono considerate le basi del calcolo parallelo. Inoltre, già l'anno precedente, cioè nel 1966 Michael J. Flynn classificava le architetture dei sistemi di calcolo secondo la capacità di elaborare contemporaneamente flussi multipli di istruzioni o dati.

Tale classificazione, nota con il nome di Tassonomia di Flynn, è ancora oggi valida ed è in grado di ripartire tutti i sistemi di calcolo in una delle seguenti categorie:

- SISD (Single Instruction Single Data)
- SIMD (Single Instruction Multiple Data)
- MISD (Multiple Instruction Single Data)
- MIMD (Multiple Instruction Multiple Data)

Di particolare interesse sono i sistemi che ricadono nelle categorie SIMD e MIMD, e rappresentano le tipologie sulle quali oggi maggiormente si concentra l'attenzione della ricerca e del mercato.

Di seguito sono presentate le caratteristiche dei principali sistemi di calcolo appartenenti a ciascuna categoria[1].

1.2.1 SISD (Single Instruction Single Data)

Le istruzioni vengono eseguite sequenzialmente su un dato per volta. Fanno ad esempio parte di questa categoria tutte le macchine a registri con architettura di von Neumann e quindi la maggior parte degli elaboratori ad uso privato oggi in commercio.

[1] Tuttavia, la natura della materia trattata, caratterizzata da avanzamenti rapidissimi, non permette una classificazione rigorosa che raccolga effettivamente tutti i sistemi di calcolo.

La maggior parte dei moderni sistemi di questa tipologia è tuttavia in grado di eseguire più istruzioni concorrentemente tramite l'utilizzo di tecniche sofisticate per la ripartizione del tempo d'utilizzo dell'unità d'elaborazione tra più sequenze di istruzioni, come ad esempio l'uso del *multi-threading* e delle *pipeline*.

1.2.2 SIMD (Single Instruction Multiple Data)

Le unità di elaborazione dei sistemi di questa categoria vengono anche denominate "processori vettoriali" in quanto ogni singola istruzione di tali sistemi opera su array di dati.

Mentre in precedenza il calcolo vettoriale era prerogativa esclusiva dei super computer (il primo di essi a sfruttare il calcolo vettoriale fu il Cray X-MP del 1982), al giorno d'oggi la maggior parte dei microprocessori multi-purpose commerciali contiene sottoinsiemi di istruzioni che operano su più dati contemporaneamente (ad esempio le istruzioni SSEx dei processori x86 Intel), così come tutte le moderne schede grafiche sono equipaggiate con GPU (Graphical Processing Unit) capaci di calcoli vettoriali e matriciali per l'utilizzo in applicazioni grafiche.

Tuttavia i più diffusi sistemi di consumo di massa di questo tipo operano su array di dimensione di gran lunga minore rispetto ai processori vettoriali impiegati nei super computer poiché sviluppati principalmente per applicazioni di tipo multimediale piuttosto che di calcolo vero e proprio.

I sistemi capaci di calcolo vettoriale implementano il così detto "parallelismo sui dati" poiché capaci di operare su più dati contemporaneamente.

A dispetto della loro efficienza nell'esecuzione di algoritmi che operano su vettori di dati, le architetture di tipo SIMD sono spesso molto complesse e necessitano di software scritto ad-hoc. È per questo ed altri motivi come l'alto costo di realizzazione, che nel panorama del calcolo ad alte prestazioni i sistemi di questo tipo sono andati via via diminuendo, lasciando sostanzialmente il posto a sistemi con architettura di tipo MIMD.

1.2.3 MISD (Multiple Instruction Single Data)

Rientrano in questa tipologia tutti quei sistemi che sono in grado di eseguire contemporaneamente più istruzioni sullo stesso dato.

Tipicamente nella maggior parte dei problemi si ha a che fare con grandi quantità di dati. Perciò le architetture di tipo SIMD e MIMD sono state impiegate in misura di gran lunga maggiore rispetto a quelle MISD. Non esistono dunque sistemi commerciali di questa tipologia che hanno conosciuto una rilevante diffusione.

1.2.4 MIMD (Multiple Instruction Multiple Data)

Contenendo più unità di elaborazione indipendenti, i sistemi con architettura MIMD sono in grado di eseguire contemporaneamente più istruzioni diverse su dati diversi e trovano il loro naturale impiego nel calcolo parallelo o in applicazioni che beneficiano del parallelismo.

Le implementazioni di sistemi che ricadono in questa categoria hanno permesso di raggiungere le più alte prestazioni fino ad ora ottenute. La lista top500.org che classifica i cinquecento più potenti super computer in produzione[2], è difatti dominata da sistemi che ricadono in questa tipologia.

La varietà di sistemi MIMD è tale da poter introdurre un'ulteriore classificazione in base all'accesso alla memoria da parte delle unità d'elaborazione:

MIMD a memoria condivisa

In questo caso tutti i processori che sono contenuti nel sistema condividono la stessa memoria per mezzo di hardware dedicato.

Le macchine che contengono da 2 a 64 unità di elaborazione indicate col nome di SMP (Simmetric Multi Processing) fanno parte di questa categoria.

Gli elaboratori SMP possono essere a loro volta distinti in base a come i vari processori che contengono accedono alla memoria principale.

Le architetture che consentono ai processori un accesso paritetico a tutta la memoria prendono il nome di UMA (Uniform Memory Access). Tuttavia la condivisione di tutta la memoria può rappresentare un potenziale "collo di bottiglia" quando i processori concorrono per l'accesso ad essa.

Nelle architetture di tipo NUMA (Non-Uniform Memory Access) invece, ogni processore, pur mantenendo una visione globale dell'intera memoria del sistema, ha una limitata area di memoria locale alla quale accede esclusivamente e con velocità maggiore, mentre si ha una perdita di prestazioni solo nel caso di accessi ad aree di memoria comuni, o comunque non locali.

In questi casi l'accesso deve essere regolato da complessi meccanismi necessari per mantenere coerente lo stato della cache di ogni processore. Ciò comporta una crescita della complessità dell'hardware al crescere del numero di processori.

Per essere realmente utilizzabili le architetture di tipo NUMA dipendono pesantemente da tali meccanismi per mantenere la coerenza, infatti ci si riferisce a tali architetture con l'acronimo ccNUMA (cache-coherent NUMA).

[2]La classifica è basata sulla valutazione attraverso il benchmark LINPACK delle capacità di calcolo media e di picco misurata in FLOPS (Floating point Operations Per Second), cioè operazioni in virgola mobile al secondo

MIMD a memoria distribuita

In questa categoria vanno inseriti tutti i sistemi che possiedono un'area di memoria separata per ogni unità di calcolo (detta anche nodo) ed ogni area di memoria non è direttamente accessibile da parte di altri nodi.

I più potenti sistemi di calcolo oggi utilizzati appartengono a questa categoria e ci si riferisce ad essi con l'acronimo MPP (Massively Parallel Processing).

Negli MPP ogni nodo contiene una o più unità di elaborazione e aree di memoria ad essi associate. Ogni nodo è connesso agli altri con topologie spesso complesse, volte ad ottimizzare costi, ampiezza di banda e latenza. L'efficienza dell'interconnessione tra i nodi è cruciale per le performance dell'intero sistema poiché non essendoci porzioni di memoria condivisa, il passaggio di dati tra le unità di elaborazioni può avvenire solo attraverso lo scambio di messaggi.

Tutti in nodi di elaborazione affiancati da altri nodi specializzati per le operazioni di input/output contribuiscono a formare un unico grande calcolatore.

Oltre ai sistemi MPP, la tipologia di sistemi MIMD a memoria distribuita che negli ultimi anni ha ricevuto maggiori attenzioni da parte delle comunità scientifiche e dai produttori di hardware è rappresentata dai *cluster di computer* (letteralmente "grappoli di computer").

Spesso i nodi degli MPP fanno uso di unità di elaborazione di tipo SIMD in modo da aumentare ulteriormente il grado di parallelismo.

Similmente agli MPP anche i cluster di computer sono costituiti da più nodi, ma a differenza dei primi, ogni nodo è costituito a sua volta da un sistema indipendente che comprende unità di elaborazione, memoria e periferiche di input-output.

Il resto del capitolo sarà dedicato agli aspetti peculiari di questi sistemi.

1.2.5 Diffusione del Cluster Computing

Il cluster computing ad alte prestazioni, inteso come l'impiego di cluster di computer nel calcolo ad alte prestazioni, ha rappresentato a partire dai primi anni '90 la vera rivoluzione del super-calcolo.

La lista top500.org, già sopra citata, è un'eloquente testimonianza della rilevanza del cluster computing nella scena odierna. La figura 1.1 illustra la distribuzione delle tipologie di architetture dei 500 sistemi più performanti del pianeta dal 1993 al 2007. La dimensione della diffusione dei cluster di computer tra i "top 500" è ben illustrata dal seguente dato: nella lista del 1998 compariva solo un cluster computer; dieci anni dopo, nella lista del

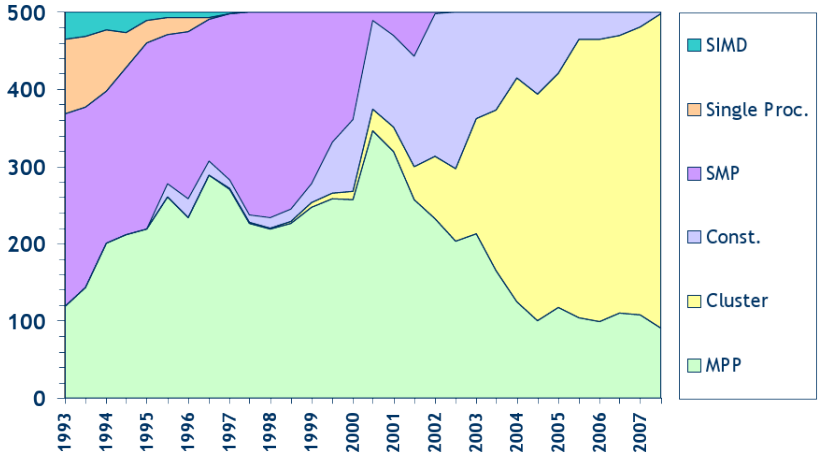


Figura 1.1: Distribuzione delle architetture utilizzate nei 500 sistemi di calcolo ad alte prestazioni censiti dalla lista top500.org dal giugno 1993 al settembre 2007.

novembre 2007, il numero di cluster è cresciuto fino a raggiungere quota 406, cioè l'81,20% dei super computer censiti.

Ma quali sono le ragioni di una così ampia diffusione? Esse sono molteplici, e sono tutte da ricercarsi tra i risultati dello sviluppo delle tecnologie informatiche raggiunti a partire dagli anni '90. Le motivazioni principali possono essere riassunte come segue:

Incremento della potenza di calcolo dei computer COTS: Si può affermare che la spinta iniziale alla larga adozione dei cluster di computer è dovuta in larga misura all'impressionante tasso di crescita della potenza di calcolo dei computer commerciali COTS e al relativo abbattimento dei prezzi. Con l'acronimo COTS (Commercial, Off-The-Shelf) si indicano tutte quelle componenti, generalmente hardware, che beneficiano della diffusione del mercato di massa, e vengono fornite agli acquirenti già pronte per l'utilizzo.

Il mercato dei Personal Computer e delle Workstation ad uso professionale ha conosciuto un forte incremento negli anni '80, e una vera e propria esplosione a partire dal decennio scorso. Il tasso di incremento dell'integrazione dei transistor nei microprocessori e della conseguente potenza di calcolo dei calcolatori, ha in molti periodi superato le previsioni fornite dalla nota Legge di Moore, raddoppiando ogni 18 mesi. Questa crescita per singola unità ha ovviamente avuto ripercussioni sull'abbattimento del rapporto costo/flop, cioè sul drammatico abbassamento degli investimenti necessari per ottenere un incremento della quantità di operazioni in virgola mobile per secondo.

Rapido sviluppo delle tecnologie di rete: Sempre negli stessi periodi nuove e più performanti soluzioni di rete hanno permesso ad un largo bacino di utenza di potersi dotare di infrastrutture di rete con ampiezze di banda sempre più alte e tempi di latenza inferiori. L'interesse crescente verso le tecnologie di rete, e l'impiego di protocolli di comunicazione standard principalmente dovuto alla diffusione esponenziale della rete globale, ha inoltre contribuito alla convergenza delle soluzioni di rete verso ambienti caratterizzati dall'alta interoperabilità, di cui i cluster possono direttamente beneficiare.

Software maturo e generalmente affidabile: Nelle architetture parallele proprietarie quali gli MPP, il software necessario all'utilizzo deve essere sviluppato appositamente e deve attraversare lunghe fasi di progettazione, realizzazione e validazione, onerose sia in termini di tempo che di costi.

Per contro, la dotazione software delle workstation e dei personal computer è diventata nel corso degli anni estremamente ampia e direttamente utilizzabile nella maggior parte delle applicazioni. Accanto ad essa la nascita del modello di sviluppo *Open Source* (a cui si devono strumenti come ad esempio GNU/Linux, il progetto Beowulf o OpenMOSIX) ha ulteriormente ampliato le possibilità di utilizzo dei sistemi COTS fornendo applicazioni ormai mature e facilmente adattabili alle specifiche esigenze.

Espandibilità e Scalabilità: I cluster di computer consentono ad organizzazioni e privati di potersi dotare di infrastrutture di calcolo dalle capacità un tempo accessibili solo a grandi enti con budget molto alti.

A fronte di un'investimento iniziale relativamente contenuto i cluster possono essere costruiti in risposta a esigenze e possibilità economiche che possono variare nel tempo. I cluster possono essere facilmente espansi o ridotti in modo da seguire queste esigenze e possibilità, adattandosi, dove richiesto, in maniera naturale alla continua crescita della capacità di calcolo dei sistemi COTS.

Come si è visto, oggi sono moltissime le implementazioni di cluster che raggiungono capacità di calcolo paragonabili a quelle di soluzioni specificatamente progettate e costruite per il calcolo ad alte prestazioni, ma con costi equivalenti a un decimo di quelli relativi alle architetture specializzate.

1.3 Definizione ed Architettura

Benché in una visione più ampia si possa affermare che una qualunque rete di computer costituisca un cluster, e che le reti di computer stesse hanno come fine ultimo quello della realizzazione di un sistema di condivisione delle risorse, in questo capitolo si farà riferimento ai cluster di computer intesi in senso stretto.

In questa accezione, un *cluster di computer* è da intendersi come

> *un sistema di calcolo distribuito, costituito da una collezione di computer indipendenti interconnessi che lavorano insieme come un'unica risorsa di calcolo integrata.*

I computer che costituiscono il cluster vengono detti *nodi* e ognuno di essi è capace di eseguire applicazioni individualmente.

L'interconnessione tra i nodi è tipicamente costituita da una o più LAN ad alta velocità ed è impiegata per il passaggio di messaggi tra i nodi e l'interfacciamento del sistema con l'esterno. L'infrastruttura di interconnessione costituisce una SAN (System Area Network), cioè una porzione di rete espressamente dedicata alla realizzazione del sistema.

A differenziare i cluster di computer intesi in questo senso dalle generiche reti di calcolatori vi è il fatto che i cluster sono progettati e realizzati come una singola entità ed in risposta ad una precisa necessità, tenendo conto di parametri di ottimizzazione per una o più applicazioni (non necessariamente il calcolo ad alte prestazioni) che ne condizionano profondamente le scelte architetturali.

1.3.1 Architettura: Il Livello Fisico

Tipicamente un cluster è una struttura locale ben definita, soggetta all'amministrazione di un unico ente e che occupa un unico ambiente atto ad ospitare il cluster stesso.

Indipendentemente dalle peculiarità delle molteplici realizzazioni fisiche, le componenti hardware fondamentali sono i nodi e la rete d'interconnessione tra di essi.

1.3.2 Nodi

Possono essere impiegati come nodi sistemi mono o multi processore quali: Personal Computer (PC), Workstation o SMP, originariamente destinati al mercato desktop o server. Essi forniscono ad un cluster capacità di calcolo e di storage. Ogni singolo nodo contiene l'hardware comunemente disponibile nei

computer oggi in commercio. In particolare ogni nodo è tipicamente dotato di:

- Un' unità d'elaborazione centrale nel caso dei PC, o anche più di una nel caso di SMP o Workstation.
- Memoria principale (RAM).
- Una o più interfacce di rete (*Fast* o *GigaBit Ethernet*).
- Memoria secondaria: Dispositivi di storage (Hard Disks, ecc.).

La configurazione dei nodi tuttavia non è necessariamente omogenea: alcuni nodi possono essere destinati a compiti specifici, come ad esempio operazioni di input/output intensive o esclusivamente di calcolo. In questi casi la dotazione hardware può essere facilmente adattata per meglio soddisfare le necessità che la particolare destinazione d'uso introduce.

I nodi sono dunque sottosistemi completamente autonomi dal punto di vista operazionale, e ciò fa sì che non vi sia la necessità di dotarli di ulteriore hardware per un uso general-purpose, potendo così direttamente beneficiare dalla notevole flessione di costi nell'hardware di largo consumo (COTS) degli ultimi anni. E' tuttavia possibile configurare i singoli nodi secondo necessità senza grosse ripercussioni sull'intera struttura.

Nelle prime implementazioni i nodi venivano ricavati da PC, Workstation o Server con il tipico formato "a torre". Successivamente, si è imposto il formato standard *1U* che consente di ridurre la superficie necessaria per numero di nodi. Grazie a questo formato i nodi possono essere impilati uno sull'altro e montati in appositi armadi detti "*rack*" che risiedono in locali adeguati dal punto di vista dell'alimentazione elettrica, condizionamento e sicurezza.

1.3.3 Rete

Sono molte e varie le tecnologie di rete oggi disponibili per la realizzazione dell'infrastruttura di interconnessione dei nodi. La scelta della componente di rete è resa di cruciale importanza da vari fattori quali i tempi di latenza, l'ampiezza di banda, i costi e, non ultima, la compatibilità tra i nodi.

È per la proprietà di rappresentare un compromesso nell'ottimizzazione di tali fattori che spesso la tipologia di rete utilizzata è basata sullo standard de facto *Ethernet*.

In particolare nella versione *GigaBit*, è in grado di offrire ampiezza di banda di 1Gbps con costi relativamente ridotti e tempi di latenza minimi dovuti solo all'overhead dei protocolli di livello superiore.

Tecnologie più costose possono tuttavia offrire vantaggi in termini di prestazioni nella comunicazione tra i nodi. E' il caso ad esempio di soluzioni quali il 10 *Gigabit Ethernet*, *FibreChannel*, o le reti *Myrinet* della Myricom (caratterizzate dai bassi tempi di latenza e da ampiezza di banda fino a 10Gbps), o ancora dalle soluzioni *InfiniBand* capaci di throughput[3] ancora maggiori.

La topologia della rete dipende fortemente dalla tecnologia di rete utilizzata. Non è tuttavia raro l'utilizzo di più tecnologie per porzioni diverse dell'infrastruttura di connessione, nella quale sono spesso identificabili aree organizzate gerarchicamente: backbone con frequenti scambi di messaggi, e che necessitano prestazioni più elevate, ed aree più periferiche con meno traffico.

1.3.4 Architettura: Caratteristiche Software

Come per ogni altro sistema di calcolo complesso, l'utilizzo efficiente dei cluster di computer è subordinato all'impiego di sovrastrutture logiche che permettano di astrarre dalla complessità dei dettagli realizzativi del sistema stesso. Inoltre la natura distribuita degli ambienti dei cluster di computer porta con se nuove problematiche inerenti la gestione delle risorse in precedenza non rilevanti per le architetture più strutturate[4].

Da un lato, va considerato che ogni nodo, essendo un sistema autonomo, ha bisogno di eseguire un'istanza indipendente del sistema operativo. Ciò implica che i singoli nodi non hanno una diretta conoscenza delle risorse disponibili nell'intero sistema.

Dall'altro lato, è auspicabile che l'intera struttura di calcolo possa essere percepita come un unico sistema dagli utenti, sgravandoli dalla necessità di conoscere i dettagli dell'architettura del sistema. Sono dunque necessari dei meccanismi che consentano l'astrazione dalla granularità introdotta impiegando più sistemi d'elaborazione indipendenti.

L'astrazione richiesta è naturalmente demandata alla componente software, che rispetto ai sistemi software tradizionali, si arricchisce di un'ulteriore strato: il *middleware*. Come illustra la figura 1.2 esso è posto immediatamente sopra i sistemi operativi dei singoli nodi e sotto le applicazioni degli utenti.

In esso si possono individuare diverse componenti che insieme contribuiscono a formare la sovrastruttura logica del cluster.

[3] ampiezza di banda effettiva

[4] Con "architetture o sistemi più strutturati" si intendono i sistemi "*tightly-coupled*", che differiscono dai cluster per il più alto grado di dipendenza tra le parti che li compongono. Ad esempio, i sistemi SMP sono più strutturati dei cluster in quanto ogni processore condivide la stessa memoria facendo dipendere da essa il funzionamento dell'intero sistema.

Nel paragrafo successivo verranno presentati gli aspetti principali e i concetti che il software più comunemente utilizzato nei cluster è preposto ad implementare.

Va tuttavia considerato che esistono molte possibilità di implementazione e di scelta del software di un cluster e che molto spesso l'intera infrastruttura software è frutto dell'integrazione di molte componenti, alcune già disponibili ed altre altamente personalizzate. Questo tipo di integrazione è resa possibile dalla natura flessibile e configurabile dei cluster che ben si combina con software a codice aperto altrettanto adattabile alle proprie particolari necessità.

La trattazione che segue è basata su considerazioni e analisi di carattere generale il cui intento è quello di descrivere le caratteristiche del software di un cluster in modo più generale possibile e non si riferisce in particolare, se non dove esplicitamente specificato, alle peculiarità di alcune implementazioni esistenti.

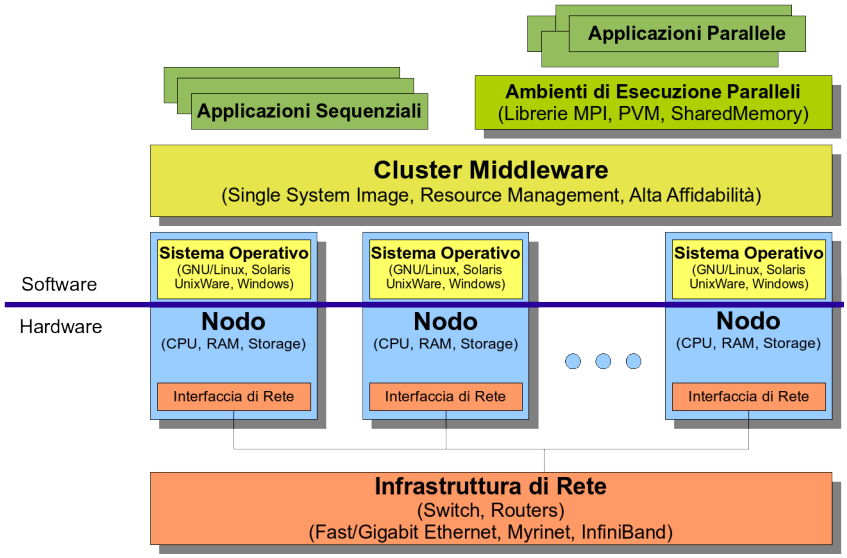


Figura 1.2: Schema ad alto livello di astrazione dell'architettura hardware e software di un cluster di computer.

1.3.5 Il Sistema Operativo

Come già accennato, ogni singolo nodo necessita di eseguire un'istanza separata del sistema operativo per la gestione a basso livello delle risorse che condivide con il resto del cluster. Ciò non sorprende, infatti ogni singolo nodo è a tutti gli effetti un sistema completamente indipendente, progettato per

essere impiegato in modo autonomo, e dunque che necessita della medesima astrazione software di qualunque altro computer di tipo workstation o server.

Il sistema operativo fornisce ai nodi tutte le funzioni basilari che ci si aspetterebbe da un comune sistema operativo: gestione della (o delle) CPU, della memoria, della rete e delle periferiche di I/O. Inoltre il sistema operativo costituisce una solida piattaforma, fatta di librerie e applicazioni di sistema, sulla quale è possibile implementare ulteriori strati di software che consentono una piena astrazione dall'hardware e una fruizione più naturale e efficiente del computer. Nella figura 1.3 è mostrato lo schema generale di un

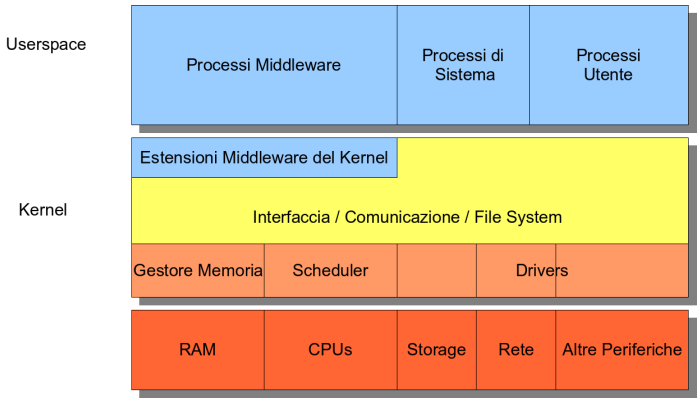


Figura 1.3: Schema a blocchi del sistema operativo di un nodo nei cluster di computer.

sistema operativo di un nodo di un cluster. In esso sono evidenziati gli strati d'astrazione e delle componenti principali del *nucleo* o *kernel* e dello strato superiore ad esso. Quest'ultimo strato rappresenta l'ambiente logico con il quale gli utenti e le applicazioni di più alto livello interagiscono. All'interno di tale ambiente, detto *userspace* vi sono anche i processi di sistema che sono a tutti gli effetti parti integranti del sistema operativo e curano aspetti fondamentali della gestione del computer.

Come la figura suggerisce, l'implementazione di alcune funzioni del middleware sono strettamente connesse alle funzioni di gestione che, a più basso livello, devono essere svolte dal kernel del sistema operativo. Molte implementazioni del middleware prevedono infatti la modifica o l'estensione del kernel del sistema operativo in esecuzione sui nodi. Queste modifiche servono a garantire alle componenti di gestione delle risorse del middleware un maggiore controllo sulle risorse ospitate dai singoli nodi, che altrimenti sarebbero governate esclusivamente dalle decisioni del kernel, prese basandosi sulla visione non globale del sistema in cui si trova. Da questo punto di vista è di primaria importanza che il sistema operativo scelto per i nodi di un cluster

sia facilmente estensibile in termini di supporto hardware e di nuove funzionalità. È anche molto importante che il sistema operativo supporti i comuni protocolli di comunicazione e gli standard nei paradigmi di programmazione (multitasking e multithreading, RPC, sockets, ecc.), per garantire una piena interoperabilità che rappresenta un fattore di fondamentale importanza in ambienti distribuiti come quello dei cluster.

È per questo motivo che sono molte le implementazioni di cluster che utilizzano sistemi operativi a sorgente aperto basati sullo standard POSIX (come ad esempio GNU/Linux, xBSD) con opportune modifiche per adattarli ad un ambiente distribuito e parallelo.

1.3.6 Middleware

Il middleware in un cluster di computer è costituito da varie componenti che possono essere implementate a vari livelli della struttura software del cluster. Il middleware ha il compito di nascondere l'eterogeneità dovuta alla natura distribuita del cluster. Ogni componente del middleware risponde a precise necessità del sistema e implementa meccanismi e infrastrutture alla base di concetti chiave per l'utilizzo efficiente del cluster:

- Efficiente gestione delle risorse di calcolo.
- Proprietà di Single System Image.
- Alta disponibilità e tolleranza ai guasti.

Benché generalmente auspicabili, la maggiore importanza di una o dell'altra proprietà introdotta dalle componenti del middleware è relativa alla destinazione d'uso del cluster, che può essere più orientata verso l'offerta di servizi, e quindi necessitare di meccanismi di "fail-over" in caso di guasti e "load balancing" in situazioni di alto traffico, o verso il calcolo ad alte prestazioni e dipendere fortemente da un'ottima gestione delle risorse.

Le componenti del middleware che arricchiscono il cluster delle proprietà elencate, contribuiscono a fornire agli utenti una struttura integrata solida e di facile utilizzo, insieme a facilitare l'amministrazione del cluster.

Management e Scheduling delle Risorse di Calcolo

Il Management e Scheduling delle Risorse (RMS), traducibile come gestione ordinata delle risorse, è di fondamentale importanza per l'uso efficiente del cluster. Il compito delle componenti di RMS è infatti quello di distribuire su tutto il cluster le richieste di risorse hardware delle applicazioni in esecuzione.

I software che svolgono funzioni di RMS consistono solitamente di due parti: un *resource manager* e un *resource scheduler*.

La componente di "resource management" è responsabile per la raccolta di informazioni relative alla presenza e allo stato delle risorse. Essa si preoccupa di mantenere un'immagine dello stato delle risorse disponibili nel cluster e provvede ad aggiornare la sua conoscenza del sistema in base ai cambiamenti dovuti ad allocazioni, deallocazioni o indisponibilità dipendenti da altre ragioni quali il malfunzionamento dell'hardware. È suo compito inoltre raccogliere e autenticare le richieste di allocazione delle risorse, insieme a creare le strutture dati necessarie all'identificazione dei processi delle applicazioni che ne richiedono l'utilizzo.

Accanto alla componente di management, quella di "resource scheduling" si interessa di ordinare le richieste in modo da minimizzare i tempi di attesa e di gestire le situazioni di concorrenza per l'utilizzo delle risorse da parte delle applicazioni. Per svolgere il suo compito deve fare riferimento alle informazioni riguardo le richieste e la disponibilità delle risorse mantenute aggiornate dalla componente di managment.

Tipicamente lo scheduling (ordinamento) delle richieste viene regolato per mezzo di code gestite con politiche il più possibile configurabili in modo da ottenere un utilizzo delle risorse efficiente e nel contempo equamente distribuito tra le applicazioni.

L'architettura tipica delle applicazioni di RMS è di tipo client/server: un processo di sistema resta in esecuzione su ogni nodo del sistema che condivide le proprie risorse, mantenendo aggiornate le informazioni relative al sistema che lo ospita e secondo protocolli e modalità stabilite, le comunica alla componente di management. Alla componente di management vengono in questo modo comunicati tutti i cambiamenti di stato delle risorse che avvengono sui singoli nodi.

L'utente ha la possibilità di sottomette le sue richieste di esecuzione di un'applicazione attraverso un client d'interfaccia che interagisce con il sistema di RMS, ed in particolare con la componente di management. Le applicazioni possono essere di tipo *interattivo* o nel caso più comune, di tipo *batch*. Nel secondo caso le applicazioni sottomesse per l'esecuzione al sistema di RMS vengono dette *job*.

Nella sottomissione di un job l'utente deve provvedere a fornire al sistema di RMS solo i dettagli necessari per l'esecuzione dell'applicazione come la posizione dell'eseguibile, i dati in ingresso, la destinazione dei dati in uscita, insieme alle proprie credenziali per l'autenticazione e autorizzazione.

In seguito alla sottomissione di una richiesta di esecuzione è il sistema di RMS che si prende cura della corretta esecuzione del job, permetten-

do agli utenti l'astrazione dalla complessità dei livelli software e hardware sottostanti.

Il panorama di software che svolge funzioni di RMS è molto ampio. Tra di essi vi è la famiglia di batch system PBS (Portable Batch System), a partire dalla quale sono stati sviluppati *OpenPBS*, *TORQUE* o *PBS Professional*. Accanto ad essi i più utilizzati sono *Maui Cluster Scheduler*, *LSF* (*Load Sharing Facility*), *Condor*, *Sun Grid Engine* e molti altri.

Single System Image

Per *Single Sistem Image* (SSI) si intende la proprietà di un sistema di nascondere la natura distribuita ed eterogenea delle risorse disponibili presentandole agli utenti come un'unica risorsa di calcolo.

Un sistema con questa proprietà consente all'utente finale di ignorare la natura distribuita del sistema, consentendo l'accesso ad esso attraverso interfacce familiari e ben delineate che definiscono i punti di ingresso del cluster e che consentono l'utilizzo delle risorse senza la necessità di conoscere su quali nodi esse siano fisicamente ospitate.

Alla luce di ciò, la componente di RMS può essere considerata parte integrante del software che contribuisce a fornire la proprietà di SSI al sistema.

Accanto ad essa vi sono le componenti che gestiscono lo storage che consentono di accedere ai dati in modo indipendente dalla reale posizione degli stessi. Queste componenti sono spesso gestite attraverso l'impiego di *filesystem distribuiti* che forniscono ad utenti ed amministratori la possibilità di accedere ai dati attraverso una singola gerarchia di file e directory, gestendo in maniera trasparente la concorrenza degli accessi e la consistenza dei dati.

Un altro aspetto legato alla proprietà di SSI e quello del *virtual networking*: gli utenti e le applicazioni possono utilizzare le connessioni di rete che afferiscono al cluster in modo non vincolato dalla reale topologia fisica della rete e indipendentemente dal fatto che una determinata rete sia direttamente collegata al nodo dove l'applicazione è in esecuzione.

Le componenti che implementano l'SSI facilitano inoltre l'amministrazione del cluster fornendo astrazioni quali un unico id per i job in esecuzione, o strumenti per la gestione degli utenti centralizzata e replicata su tutti i nodi.

Tra le implementazioni di software che forniscono proprietà di SSI vanno ricordati *OpenSSI*, *Kerrighed*, i pacchetti software *OpenMOSIX* e quelli forniti dai sistemi *Beowulf*.

Alta Disponibilità ed Alta Affidabilità

Le componenti middleware vengono anche impiegate per risolvere o limitare i danni dovuti ai guasti hardware o ai malfunzionamenti del software. Quando questi malfunzionamenti sono di particolare gravità devono poter essere arginati attraverso meccanismi che permettano al cluster di continuare ad operare fino all'intervento dei tecnici.

L'alta disponibilità e l'alta affidabilità riguardano le proprietà di un sistema di essere tollerante ai guasti senza interrompere la possibilità di fruizione dei servizi offerti.

Sebbene in sistemi complessi come i cluster la varietà dei guasti che si possono presentare è decisamente ampia e oggetto di altrettanto ampi studi, in questa sede considereremo solo due dei principali effetti negativi che un guasto può determinare:

- interruzione dei job in esecuzione e dei servizi offerti,
- perdita del contenuto informativo del cluster.

Nel primo caso è auspicabile che i risultati parziali ottenuti fino all'interruzione del job non vengano perduti, o che l'intero processo possa riprendere immediatamente facendolo migrare su altri nodi. Allo stesso modo un servizio in esecuzione sul cluster deve poter essere ripristinato nel più breve tempo possibile.

Nel secondo caso, quando un guasto determina la corruzione o la perdita di dati archiviati nei dispositivi di storage del cluster, è importante avere la possibilità di ricostruire le informazioni perdute o corrotte.

In un cluster con proprietà di alta affidabilità e disponibilità, l'utilizzo di un certo grado di ridondanza, insieme a tecniche che osservano lo stato di processi e risorse, permette nella maggioranza delle evenienze di risolvere i problemi sia nel primo che nel secondo caso.

I job costretti all'arresto possono contare su strumenti quali il *checkpointing* che consiste nel salvare periodicamente i parametri necessari alla ricostruzione del job, e consente alle componenti di alta affidabilità di ripristinare i job dall'ultimo stato salvato prima dell'interruzione.

Il cluster è inoltre protetto dalla perdita di dati da strumenti che si preoccupano di introdurre e gestire la ridondanza dei dati con sistemi RAID e di backup.

Inoltre tecniche di *fail-over* e *mirroring* arginano l'indisponibilità dei servizi attraverso il bilanciamento del traffico su più macchine preposte all'offerta dello stesso servizio.

In quest'ultimo aspetto ci si può affidare a tecniche di virtualizzazione che verranno esposte in dettaglio nel prosieguo del successivo capitolo. Tramite la virtualizzazione, i servizi di primaria importanza detti "*mission critical*", possono essere offerti su macchine virtuali che costituiscono una valida alternativa alla replicazione in hardware.

1.3.7 Ambienti d'Esecuzione

In cima alla struttura software vi sono le componenti che supportano l'esecuzione delle applicazioni degli utenti. La disponibilità di software standard per la programmazione e l'esecuzione di applicazioni parallele ha aiutato notevolmente la diffusione dei cluster di computer.

Un'applicazione parallela esegue un programma il cui carico computazionale può essere suddiviso tra i vari nodi, consentendo vantaggi prestazionali proporzionali al grado di parallelismo del programma stesso.

I cluster di computer sono in grado di offrire ambienti di esecuzione per le applicazioni parallele tramite l'impiego di librerie che "parallelizzano" i programmi. È il caso delle librerie che consentono di sfruttare il paradigma di programmazione *message passing*. Le librerie di questo tipo forniscono meccanismi per formare canali di comunicazione e consentire il passaggio di messaggi esplicito tra i processi che compongono le applicazioni parallele.

Le specifiche di *Message Passing Interface* (MPI) costituiscono ormai uno standard de facto per la programmazione di applicazioni parallele in sistemi a memoria distribuita come i cluster, sebbene possano essere usate anche in sistemi più strutturati.

Esistono molte implementazioni di librerie MPI e la standardizzazione promossa in primo luogo dall'*MPI Forum* ha aumentato la portabilità sia delle librerie che delle applicazioni che ne fanno uso. Esse sono ormai disponibili per un gran numero di architetture hardware e possiedono interfacce verso tutti i linguaggi di programmazione più utilizzati.

Accanto alle librerie MPI, le librerie di message passing più utilizzate sono le librerie *Parallel Virtual Machine* (PVM).

Oltre alle librerie di message passing un altro approccio è quello adottato dalle librerie *Distribuited Shared Memory* (DSM) che forniscono un'area di memoria condivisa virtuale ai processi paralleli.

Sebbene la programmazione sia resa più semplice da questo secondo approccio, le caratteristiche di scambi di messaggi espliciti di MPI consente una maggiore granularità nella gestione dell'utilizzo della rete, che spesso le fa preferire. L'insieme di librerie di cui si è discusso costituiscono l'ambiente d'esecuzione parallelo riportato in figura 1.2.

Le librerie devono essere messe a disposizione dai sistemi operativi installati sui nodi. Sebbene ciò non rappresenti un problema in cluster di piccole dimensioni e con basso tasso di eterogeneità, in cluster di dimensioni più grandi, dotare tutti i sistemi operativi di ogni nodo del corredo di librerie necessario a soddisfare i bisogni di un largo gruppo di utenza può presentare notevoli difficoltà di gestione per via delle incompatibilità che si possono presentare.

Questo aspetto verrà approfondito nel corso dei successivi capitoli.

1.4 Dai Cluster al Grid Computing

Come si è visto il cluster computing rappresenta una valida alternativa all'impiego di architetture di gran lunga più costose. Tuttavia non è possibile considerare il cluster computing come la risposta definitiva alla necessità di prestazioni sempre più elevate.

In questo paragrafo si vogliono presentare i principali limiti nei quali incorrono le implementazioni di cluster di computer. Si vedrà inoltre che i tentativi di superamento di tali limiti hanno portato alla definizione di un nuovo trend: il *Grid Computing*

1.4.1 Limiti del Cluster Computing

Il cluster computing è oramai una realtà consolidata nel calcolo parallelo. Sono molte le implementazioni già esistenti, e l'architettura a cluster rappresenta lo stato dell'arte per una gamma molto ampia di applicazioni. Vi sono produttori di hardware che commercializzano soluzioni complete per il cluster computing.

Una tale diffusione ha contribuito a mostrare quelli che sono i limiti di applicazione. Eccone alcuni:

Efficenza ridotta in alcuni ambiti: In applicazioni parallele che dipendono da una frequenza di comunicazione elevata tra i sotto-processi, i cluster di computer sono svantaggiati rispetto ad architetture più strutturate. Il tempo impiegato per il passaggio di messaggi tra i processi in esecuzione sui nodi è di diversi ordini di grandezza maggiore rispetto a quello necessario in architetture quali SMP, NUMA o MPP[5]. Ciò dipende dal fatto che le architetture citate gestiscono la comunicazione a livello hardware, direttamente con la condivisione della memoria o strutture dedicate, mentre nei cluster ogni messaggio deve attraversare diversi strati software prima di essere inviato e altrettanti strati nella ricezione.

Alti oneri nella gestione: Le implementazioni di cluster molto grandi richiedono sforzi di manutenzione e gestione che spesso superano le possibilità economiche e di risorse umane dell'ente proprietario del cluster. Ciò è spiegabile attraverso la regola intuitiva secondo la quale, al crescere del numero delle componenti del sistema, diminuisce proporzionalmente il tempo medio tra i guasti. Ovvero, più sono i nodi, più frequenti saranno i malfunzionamenti.

[5]Si è accentato a tali architetture nella classificazione dei sistemi di calcolo del primo paragrafo.

Scalabilità limitata oltre una certa soglia: Vi sono alcuni fattori che sono inizialmente trascurabili per cluster di dimensioni medio-piccole ma che costituiscono seri problemi superate certe soglie. L'ingombro dei nodi e degli apparati ad essi afferenti cresce notevolmente in proporzione al numero dei nodi e quindi alla potenza di calcolo.

Ad esempio l'aumento del numero di nodi porta con se il problema dell'incremento del consumo energetico che può superare la possibilità di fornitura della rete elettrica, insieme a incidere fortemente in termini di costi.

Il consumo energetico porta a sua volta un'altrettanto alta dissipazione termica che costringe a dotare gli ambienti che ospitano il cluster di costosi sistemi di condizionamento incidendo ulteriormente sui consumi.

Eterogeneità crescente di hardware e software: L'eterogeneità in un cluster può essere introdotta a vari livelli. Ogni espansione per il potenziamento dell'hardware del cluster contribuisce ad aumentare progressivamente l'eterogeneità fisica dei i nodi e degli apparati di rete.

Insieme ad essa, in cluster condivisi da varie organizzazioni ed utenti, sopraggiunge un'eterogeneità a livello software dettata dalle necessità di installare sui nodi differenti applicazioni e pacchetti software di cui gli utenti fanno richiesta.

Spesso conciliare l'eterogeneità in modo da preservare la funzionalità del cluster insieme a soddisfare le esigenze dell'utenza può porre problemi di incompatibilità non facili da risolvere.

1.4.2 Grid Computing

Il Grid Computing rappresenta l'attuale frontiera negli sforzi volti ad ottenere prestazioni sempre più elevate attraverso la condivisione delle risorse.

Proprio come si definisce il concetto di cluster per superare la limitazione dei singoli sistemi di calcolo, nel Grid computing si definisce una nuova entità per superare i limiti esposti nel paragrafo precedente: questa entità è la *griglia computazionale* o *Grid*.

Ian Foster[6], uno dei principali promotori del concetto di Grid computing, fornisce una definizione di Grid in tre punti:

[6]Ian Foster nel 1995 fu a capo del progetto I-WAY il cui scopo era quello di condividere le risorse di calcolo di 17 centri di ricerca degli Stati Uniti. È inoltre insieme a Kesselman autore del libro "The Grid: Blueprint for a new computing infrastructure", e fondatore della "Globus Alliance", entrambi di fondamentale importanza nella successiva diffusione dell'idea di Grid.

Una grid è un sistema il quale:

1. *coordina l'utilizzo di risorse non soggette ad un controllo centralizzato*
2. *utilizzando protocolli e interfacce standard, aperte e general-purpose,*
3. *per fornire diverse tipologie di servizi rilevanti.*

Con "risorse" ci si riferisce ad ogni tipo di risorsa di calcolo o di storage, siano esse singoli PC, mainframe o cluster. Esse vengono messe a disposizione e gestite localmente da enti geograficamente anche molto distanti tra loro.

Gli enti che partecipano alla costituzione della Grid vengono dette *Virtual Organizations* (VO), e sottostanno a politiche di autorizzazione e autenticazione concordate insieme agli organi di coordinamento della Grid.

La distanza geografica fa sì che nella maggioranza dei casi la connessione tra le risorse avviene tramite Internet o reti WAN.

I protocolli impiegati *devono* essere aperti e standard e devono essere consoni a supportare un'ampia varietà di offerta di servizi.

L'autorità in questo campo è costituita dalla *Globus Alliance*, un organismo che promuove l'utilizzo delle tecnologie grid e sviluppa attivamente il pacchetto software *Globus Toolkit* sul quale sono basate molte delle implementazioni di grid.

Il calcolo in grid si presta molto bene alla risoluzione di problemi con set di dati molto grandi, ingestibili da un singolo sistema di calcolo e ad elevato grado di parallelismo. Questo tipo di problemi è divisibile facilmente in task indipendenti l'uno dall'altro, che non necessitano di alte prestazioni nello scambio di messaggi.

Il problema dell'alta latenza nella comunicazione è in definitiva l'unica limitazione delle grid. Per contro, possono essere espanse in termini di potenza di calcolo e capacità di storage virtualmente all'infinito, senza incorrere nei problemi esposti nel paragrafo precedente, tipici dei sistemi centralizzati.

L'architettura di una grid può essere anch'essa suddivisa su vari livelli come riportato in figura 1.4:

- Lo strato superiore è quello *applicativo* ed è appunto costituito dagli applicativi degli utenti. Essi devono essere opportunamente modificati per funzionare in un ambiente Grid.
- Vi è poi uno strato *middleware*[7] analogo a quello di un cluster e che fornisce funzioni di RMS, SSI e Alta affidabilità. Come per i cluster

[7]Nei cluster esso nasconde l'eterogeneità limitatamente ai singoli nodi, mentre nelle grid ciò deve avvenire su scala potenzialmente mondiale.

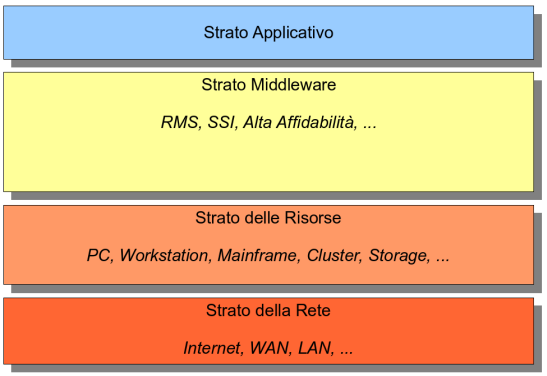


Figura 1.4: Schema ad alto livello d'astrazione di una griglia computazionale.

il middleware è di fondamentale importanza per la costituzione della grid.[8].

- Lo strato delle *risorse* contiene tutte le risorse di calcolo, storage o altro che afferiscono alla grid.
- Infine, l'ultimo strato, quello della *rete* rappresenta gli apparati che garantiscono la connettività a tutte le componenti del livello superiore.

Gli obiettivi più ambiziosi delle grid sono ancora in parte lontani dal raggiungimento. Tuttavia la diffusione di sistemi ad alte performance come i cluster, e il costante aumento delle capacità delle reti, inducono a sperare che la previsione del 1969 di Leonard Kleinrock possa un giorno diventare realtà: Kleinrock presagì la possibilità che in un futuro prossimo sarebbe stata possibile la realizzazione di un'infrastruttura di calcolo mondiale, con estensione e facilità d'uso analoghe a quelle delle odierne reti telefoniche o dell'energia elettrica.

1.4.3 LHC Computing Grid (LCG)

L'implementazione più importante e di dimensioni maggiori a livello europeo e mondiale è la grid LCG del CERN. Essa permetterà a più di 500 istituti e

[8]questi concetti sono stati esposti nel paragrafo 1.3.6.

centri di ricerca in tutto il mondo di accedere e di analizzare i dati prodotti dagli esperimenti afferenti all'*LHC* (*Large Hadron Collider*), il più grande acceleratore di particelle che sia mai stato costruito.

Si stima che la quantità di dati prodotta sarà dell'ordine di decine di petabyte all'anno, a partire dalla fine del 2008 e per i successivi 15 di attività previsti.

LCG ha una struttura gerarchica a più *tier* e si affida a realtà regionali o nazionali già esistenti per creare un unica infrastruttura di gestione e analisi dei dati su scala mondiale.

La porzione italiana di LCG è affidata all'*INFN-Grid*, ovvero la griglia computazionale compartecipata dagli istituti che ospitano le sedi dell'istituto nazionale di fisica nucleare, connessi attraverso la rete GARR.

Il progetto LCG prevede inoltre lo sviluppo di prodotti software che facilitino l'impiego delle tecnologie Grid come il middleware *gLite*.

1.5 Il Cluster INFN di Perugia

Si vuole ora presentare un caso specifico di un'implementazione di cluster computing, ed in particolare il cluster in funzione presso la sede INFN della sezione di Perugia ospitato presso il Dipartimento di Fisica.

1.5.1 Costituzione e Ingresso in INFN-Grid

Prima della costituzione del cluster, i vari gruppi di ricerca all'interno della struttura provvedevano in maniera autonoma alle necessità di strumenti di calcolo. Tuttavia questa frammentazione si rifletteva simmetricamente sugli sforzi e i costi volti alla gestione e alla manutenzione. Ciò comportava inoltre un impiego non ottimale delle risorse: esse venivano sfruttate intensamente in periodi di necessità e giacevano inutilizzate in altri.

L'accentramento in un cluster condiviso ha condotto ad evidenti vantaggi. In primo luogo ha permesso un utilizzo più intelligente delle risorse. I periodi di scarso impiego delle proprie risorse da parte di un determinato gruppo, sono divenuti sfruttabili da altri per ridurre notevolmente il tempo di calcolo e ammortizzare più rapidamente gli investimenti fatti.

Inoltre, tutti i gruppi hanno potuto beneficiare di una gestione centralizzata che ha portato ad un abbattimento dei costi in termini sia di risorse umane, sia di finanze e spazi occupati.

Per una corretta distribuzione tra i gruppi delle risorse condivise, la scelta naturale è stata quella di dotare il cluster di una componente di RMS di tipo batch (TORQUE e Maui).

Il cluster nel 2004 è poi entrato a far parte della griglia computazionale nazionale dell'INFN. Ciò ha richiesto che parte delle risorse fossero destinate all'utilizzo in grid. Ad oggi il cluster INFN di Perugia offre più di 100 CPU all'INFN-Grid ed occupa un buon livello nella Grid-INFN in produzione, contribuendo ad offrire supporto di calcolo e storage a tutti gli esperimenti afferenti all'LCG.

In vista dell'entrata in funzione dell'acceleratore LHC, il cluster sta attraversando un periodo volto all'ulteriore potenziamento degli apparati di calcolo e storage.

Dal punto di vista della dotazione software il cluster può essere definito molto eterogeneo. Per ovviare alla complessità di manutenzione e nel contempo offrire all'utenza gli strumenti di cui necessita, nel cluster sono impiegate tecniche di virtualizzazione che verranno meglio esposte a breve.

1.5.2 Architettura

Dal punto di vista hardware il cluster è costituito dai nodi e da una rete di interconnessione tra di essi. In questo paragrafo saranno esposte le caratteristiche hardware e software del cluster INFN di Perugia. Uno schema dell'architettura è mostrato in figura 1.5

I Nodi

Si può applicare una ripartizione logica dei nodi che costituiscono il cluster in dipendenza dalla destinazione d'uso. Si possono innanzitutto distinguere *nodi di frontiera* e *nodi interni*.

I nodi di frontiera sono atti ad ospitare le funzioni di interfacciamento verso l'esterno e sono gli unici accessibili all'utenza. È attraverso di essi che vengono implementate le caratteristiche di SSI del cluster.

Molti di essi sono dedicati esclusivamente a funzioni specifiche richieste dal middleware di INFN-Grid. Questi nodi sono normalmente equipaggiati con due interfacce di rete per fungere da tramite tra l'esterno e la rete interna del cluster.

Di seguito sono riportate le tipologie di nodi di frontiera individuabili nel cluster:

Computing Element (CE): Come nella maggior parte dei cluster di dimensioni medio-grandi, nel cluster INFN di Perugia vi è un solo nodo di questo tipo. Esso riveste un ruolo di primaria importanza in quanto ospita i servizi di resource management e scheduling di cui si è parlato nel paragrafo 1.3.6, relativamente alle componenti di RMS. Esso è

infatti adibito a gestire e "schedulare" le richieste d'esecuzione provenienti dall'utenza locale o Grid che vengono sottomesse per mezzo dei nodi di interfaccia.

User Interface (UI): Costituiscono gli unici nodi accessibili direttamente dall'esterno del cluster. I nodi di UI sono il mezzo attraverso cui gli utenti possono sottomettere i propri job in modo interattivo.

Normalmente le UI non sono affacciate sulla rete interna del cluster, ma nel caso in esame ciò è utile all'utenza locale poichè attraverso il login esse possono essere anche impiegate come piattaforme interattive per il test e lo sviluppo delle applicazioni.

Possono essere inoltre utilizzate per il reperimento di informazioni riguardo l'esecuzione dei propri job, dello stato delle code e per accedere ai dati ospitati dai nodi adibiti allo storage.

Storage Element (SE): L'insieme dei nodi di storage costituisce un sistema di memoria di massa utilizzabile attraverso tutto il cluster.

La somma della capacità di memorizzazione nel cluster supera i 30TB ed è in costante aumento per seguire le necessità d'archiviazione.

L'utenza che utilizza lo storage lo fa attraverso le UI che a loro volta si rivolgono agli SE e, attraverso meccanismi che gestiscono automaticamente la gerarchia di file e directory, permettono l'accesso ai dati. Esso avviene in maniera trasparente per l'utente e indipendentemente dalla reale posizione fisica.

Tipicamente molte applicazioni Grid necessitano di nodi di questo tipo.

Install Server (IS): Per semplificare l'aggiornamento e l'installazione dei sistemi operativi, nel cluster vi sono alcuni nodi adibiti a "repository" di software che consentono di effettuare l'installazione e la configurazione via rete.

Gli IS contengono in particolare l'immagine del sistema operativo necessario ai nodi da utilizzare in Grid.

L'installazione manuale risulta molto difficoltosa e gli IS facilitano enormemente la gestione del software dei nodi.

A differenza dei nodi di frontiera, quelli interni non sono direttamente visibili dall'esterno. Essi costituiscono le risorse di calcolo e di storage che compongono il cluster. Sebbene vi siano solo due tipologie di nodi interni, essi rappresentano la maggioranza dei nodi che compongono il cluster.

Le tipologie sono:

Worker Node (WN): Sono i nodi più diffusi nel cluster e il loro compito è quello di eseguire sulle proprie risorse i job sottomessi dagli utenti.

File Server (FS): I nodi di questo tipo sono per la maggior parte file server equipaggiati con sistemi RAID per la gestione di array di dischi che permettono l'archiviazione di grandi quantità di dati (dell'ordine delle decine di terabyte).

Da diverso tempo nel cluster si stanno sperimentando tecniche di virtualizzazione[9].

Questo aspetto costituisce una peculiarità del cluster INFN di Perugia e permette di arricchire la classificazione logica fin'ora esposta con un altro tipo di nodo:

Dom 0: Sono nodi atti ad ospitare un numero variabile di *macchine virtuali* (**Dom U**). Ogni macchina virtuale può rappresentare un'astrazione logica di una qualunque tipologia di nodo precedentemente esposta con l'unica limitazione dovuta alle risorse hardware della macchina Dom 0. Dopo una estensiva fase di test, sono ormai parecchi i nodi "virtualizzati" che implementano servizi importanti[10] nel cluster.

I cluster, come si è avuto modo di accennare nei precedenti paragrafi, sono sistemi che evolvono nel tempo, che si espandono secondo necessità e per seguire gli avanzamenti tecnologici.

Il cluster INFN di Perugia non fa eccezione in questo: Le macchine impiegate come nodi presentano caratteristiche molto eterogenee che riflettono il percorso evolutivo del cluster. I nodi che costituivano la maggioranza, equipaggiati con due processori Intel Pentium III ad 1Ghz sono andati via via diminuendo lasciando il posto a nodi più recenti con processori AMD Opteron64 e ancor più recentemente Intel Xeon Quad Core.

La Rete

L'infrastruttura di connessione tra i nodi è una LAN a commutazione di pacchetto di tipo Ethernet con topologia a stella a due livelli.

Il primo livello è costituito da un *"main switch"* dal quale si diramano le connessioni che raggiungono gli switch di livello inferiore. La connettività ai nodi può essere fornita sia dal main switch che dagli switch di secondo livello.

[9] Il principale sistemi di virtualizzazione impiegato è Xen, dal quale sono stati mutuati i termini usati per indicare i nodi che ospitano le macchine virtuali e le macchine virtuali stesse.

[10] Molte UI sono virtualizzate.

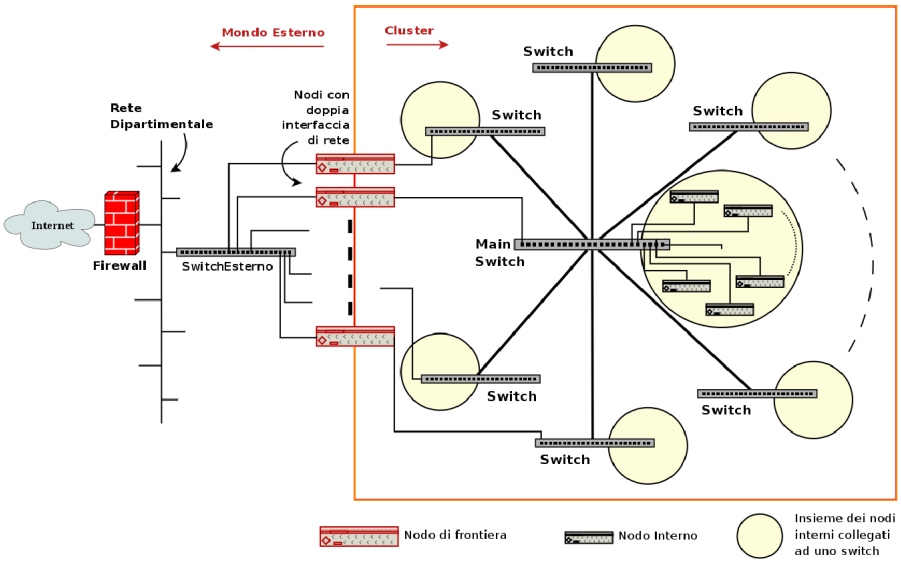


Figura 1.5: Topologia della rete del cluster INFN di Perugia.

I criteri per stabilire le connessioni rispondono a considerazioni basate sul bilanciamento del traffico, sulla disposizione fisica delle macchine e sulle capacità di connettività delle interfacce di rete di cui i nodi dispongono.

I nodi di frontiera che contengono due interfacce di rete consentono alla rete interna del cluster di poter essere affacciata sulla rete dipartimentale e quindi su internet. Le politiche d'accesso al cluster sono gestite tramite il firewall dipartimentale.

1.5.3 Peculiarità e Problematiche Emergenti

Sebbene i vantaggi offerti dalla costituzione del cluster siano evidenti e notevoli, sono sopraggiunte nel corso del periodo d'utilizzo, delle complicazioni relative alla sua gestione. Parte di esse sono dovute all'alto grado di eterogeneità sia dell'hardware che dei gruppi che hanno compartecipato alla realizzazione della struttura di calcolo. Altre problematiche dipendono dalla natura dei cluster di computer stessi che nel lungo periodo si manifestano in vari modi[11].

Di seguito vengono esposte le principali problematiche emergenti nel cluster INFN di Perugia che hanno portato alla definizione di particolari soluzioni per risolverle. Queste soluzioni rappresentano alcune delle caratteristiche peculiari del cluster.

[11]Le limitazioni in cui i cluster incorrono sono già state esposte in modo più generale nel paragrafo 1.4.1.

Diritti d'Utilizzo

Come si è già avuto modo d'esporre, il cluster INFN di Perugia nasce dalla condivisione delle proprie risorse hardware da parte di vari gruppi di ricerca concentrati nel Dipartimento di Fisica. È importante che tali gruppi non perdano la possibilità di disporre delle proprie risorse come ritengono opportuno. Tuttavia le legittime pretese avanzate dai singoli gruppi in fatto di prestazioni attese, mal si conciliano con la natura condivisa delle risorse stesse.

Nel corso degli anni d'operatività del cluster è stata posta una crescente attenzione riguardo le politiche di gestione e ripartizione delle risorse. In particolare la componente di RMS basata sul batch system *TORQUE/Maui*, è stata oggetto di parecchi studi volti ad ottimizzare le complesse politiche di scheduling necessarie ad una corretta ripartizione delle risorse.

Caratteristiche Ibride

In ambienti condivisi, ed in particolar modo nell'ambiente INFN-Grid di cui il cluster fa parte, le necessità dei singoli gruppi sono a volte in contrasto con le politiche di gestione centralizzate che caratterizzano gli ambienti Grid.

Ad esempio, il sistema operativo da installare sui nodi richiesto da INFN-Grid è Scientific Linux, ciò si traduce in una limitazione del legittimo potere decisionale dei gruppi sulle proprie risorse.

Per questo motivo è stata data la possibilità ai gruppi di decidere autonomamente la percentuale di hardware da dedicare all'impiego in Grid e all'uso locale.

L'effetto immediato di una tale decisione è quello di una "ibridazione" del cluster. Infatti l'ingresso in Grid di un cluster di computer è solitamente totale e ne presuppone un utilizzo esclusivo. Nel nostro caso invece l'accesso alle risorse può essere garantito sia ai gruppi locali che all'utenza Grid.

Per fare ciò è stato necessario fornire il cluster di più punti d'ingresso. Tipicamente essi sono rappresentati dalle UI ospitate dai nodi. Ogni gruppo necessita di una propria UI e oltre ad esse vi sono i nodi necessari all'utenza grid (il CE, diversi SE, ecc.).

Ne consegue un aumento delle macchine necessarie all'esecuzione dei servizi richiesti e una accresciuta complessità nella componente di RMS. Quest'ultima deve infatti essere configurata in modo da accettare richieste di esecuzione dei job sia dall'utenza grid che da quella locale.

Problemi Logistici

Uno dei problemi principali che il cluster si sta trovando ad affrontare in seguito alla crescita in dimensioni è legato alla disposizione fisica delle macchine.

Disporre le macchine in modo organico e funzionale è diventato via via un problema sempre più difficile da risolvere. Inoltre l'aumento del numero delle macchine ha portato con se il problema dell'approvvigionamento energetico e dell'alta dissipazione termica.

In particolare il problema della dissipazione termica sta raggiungendo proporzioni importanti e presto saranno necessari interventi per potenziare l'attuale sistema di condizionamento.

Nel 2008 è previsto il trasferimento del cluster in locali adeguatamente attrezzati ad ospitarlo che consentirà una migliore organizzazione del cluster ottimizzando i costi necessari alla sua manutenzione.

Rete Nascosta e VLAN

Un'altra peculiarità del cluster riguarda alcuni aspetti dell'infrastruttura di rete.

Molti dei WN sono posti su una rete privata nascosta per venire incontro alla carenza di IP pubblici di cui è afflitta la sede INFN di Perugia. Le macchine facenti parte dell'infrastruttura di INFN-Grid necessitavano infatti di un indirizzo IP pubblico e l'introduzione della rete nascosta insieme ad alcune modifiche del middleware INFN-Grid hanno permesso un notevole risparmio di indirizzi.

La soluzione impiegata è stata poi mutuata dal middleware ufficiale rilasciato da INFN a partire dalla versione 2.6.0 e viene correntemente utilizzato in tutti i siti INFN-Grid.

La recente ristrutturazione della rete del dipartimento di fisica ha permesso di dotare il cluster di una particolare tecnologia di rete che fa uso dello standard IEEE 802.11Q conosciuto con il nome di Virtual LAN (VLAN). Attraverso di essa si possono creare delle reti locali virtuali che consentono di collegare tra loro macchine appartenenti a segmenti di rete diversi come se fossero sulla medesima sotto rete locale. In questo modo, si può virtualmente associare alla rete nascosta del cluster ogni computer raggiunto dalla rete dipartimentale.

In quest'ottica si potrebbero sfruttare i periodi in cui i computer dei laboratori e dei docenti non vengono utilizzati per aumentare ulteriormente la potenza di calcolo del cluster.

Macchine Virtuali

Per rispondere alla necessità crescente di macchine dedicate a precisi scopi, nel cluster sono state introdotte tecniche di virtualizzazione. In particolare, la virtualizzazione consente di ospitare più macchine virtuali sulla stessa macchina fisica, riducendo l'ingombro fisico degli apparati di calcolo, il consumo energetico e la quantità di calore dissipato.

La tecnologia utilizzata è quella fornita dal software Xen, inizialmente sviluppato presso l'università di Cambridge.

Molti servizi in produzione sono già stati migrati verso piattaforme virtualizzate. Ciò ha consentito l'implementazione di meccanismi di alta affidabilità per i servizi più importanti e ne ha semplificato la gestione.

La tecniche di virtualizzazione introdotte, a fronte di una perdita di performance trascurabile nella maggior parte dei casi, offrono altri vantaggi:

- Gli utenti possono disporre trasparentemente di macchine con sistemi operativi differenti e arbitrariamente scelti in base alle loro necessità superando le limitazioni imposte dalle politiche di INFN-Grid. L'impatto dell'eterogeneità dell'hardware viene in larga misura ridotta poiché le macchine virtuali possono essere completamente astratte da quelle reali.

- Si può disporre di *ambienti virtuali*[12] che possono essere usati per offrire servizi rapidamente implementati e smantellati altrettanto velocemente, senza influire sul normale funzionamento delle macchine fisiche.

- Applicazioni in fase di sviluppo possono essere testate in ambienti appositamente costruiti evitando di sottoporre le macchine fisiche a rischi di sicurezza derivanti da malfunzionamenti del software.

Sebbene Xen offra la possibilità di facilitare la gestione delle macchine svincolandole dallo strato fisico, il controllo delle macchine virtuali è spesso difficoltoso in ambienti come quello dei cluster dove non si ha accesso diretto alle macchine reali e si necessità di ambienti che devono essere frequentemente attivati e distrutti.

Il prototipo oggetto di questa tesi affronta questo aspetto proponendo una gestione dinamica di tali ambienti. Prima di esporre l'idea alla base e i dettagli del prototipo realizzato, nel prossimo capitolo saranno affrontati gli aspetti teorici della virtualizzazione e le caratteristiche del sistema Xen.

[12] Si tornerà sul concetto di "ambiente virtuale" nei prossimi capitoli.

Capitolo 2

Virtualizzazione

2.1 Introduzione

Durante l'era dei *Mainframe*[1] l'IBM introdusse l'idea di eseguire su di essi piï¿½ istanze del sistema operativo per ridurre gli infruttuosi tempi di inattivitï¿½ delle sue macchine e offrire all'utenza l'illusione di adoperare piï¿½ macchine tra loro separate e indipendenti[2].

Nasceva cosï¿½ l'idea di *virtualizzazione dei sistemi*: l'impiego di *macchine virtuali* indipendenti che condividono lo stesso sistema hardware. In questo capitolo si discuteranno i concetti principali della virtualizzazione.

2.2 Macchine Virtuali e VMM

In questo paragrafo viene fornita una prima definizione di macchina virtuale con riferimento all'analisi relativa ai requisiti che le architetture di sistemi di calcolo devono possedere per essere virtualizzati in modo efficiente.

Tale analisi fu condotta da G. Popek e P. Goldberg negli anni '70 ed era basata su una formalizzazione dei sistemi di calcolo di terza generazione, ossia costruiti con i primi circuiti integrati. Benchï¿½ la realizzazione dei sistemi odierni con microprocessori ad alto grado di integrazione faccia di essi computer di quarta generazione, la formalizzazione introdotta ï¿½ abbastanza generale da poter essere valida per i sistemi moderni.

[1] I Mainframe sono computer di grandi dimensioni. Durante gli anni '60 e '70 costituivano l'unica possibilitï¿½ per le grandi organizzazioni di dotarsi di strumenti di calcolo automatico. I loro costi erano e sono generalmente molto alti e il ritorno economico ï¿½ ragionevole solo se il tasso di utilizzo ï¿½ altrettanto elevato.

[2] L'IBM System/360 introdotto nel 1966, fu il primo mainframe ad introdurre tecniche di virtualizzazione.

2.2.1 Definizioni

La prima definizione di macchina virtuale venne fornita in un articolo di Gerald J. Popek e Robert P. Goldberg del 1974 attraverso l'introduzione di un *Virtual Machine Monitor* (VMM).

Il VMM viene definito come un software in esecuzione sulla macchina reale che ha il completo controllo delle risorse hardware da essa fornite. Esso crea degli *ambienti d'esecuzione* che forniscono agli utenti l'illusione di un accesso diretto alle risorse della macchina fisica. Tali ambienti d'esecuzione vengono detti *Macchine Virtuali* (VM).

La definizione formulata da Popek e Goldberg ï¿½ dunque la seguente:

> *Una macchina virtuale ï¿½ un duplicato software di un computer reale nel quale un sottoinsieme statisticamente dominante di istruzioni del processore virtuale viene eseguito nativamente sul processore fisico.*

Questa definizione esclude i sistemi di emulazione e gli interpreti software. Essi infatti sono costruiti in modo da simulare in software tutte le istruzioni della macchina virtuale e ciï¿½ comporta che ogni istruzione eseguita sull'hardware simulato deve essere tradotta per essere eseguita sulla macchina reale. Ciï¿½ contrasta con la definizione in quanto la maggior parte delle istruzioni, ovvero un sottoinsieme statisticamente dominante di esse, deve essere eseguito direttamente dal processore.

2.2.2 Il Virtual Machine Monitor

Il VMM amministra le risorse reali della macchina fisica, esportandole alle macchine virtuali. Esso deve possedere tre caratteristiche principali:

1. **Equivalenza**: Gli effetti dell'esecuzione di un programma attraverso il VMM devono essere identici a quelli dello stesso programma eseguito direttamente sulla macchina originale, ad eccezione al piï¿½ del tempo d'esecuzione dovuto all'overhead del VMM e alla ridotta disponibilitï¿½ di risorse.

2. **Efficienza**: L'ambiente virtuale deve essere efficiente. La maggior parte delle istruzioni eseguite all'interno di tali ambienti deve essere eseguita direttamente dal processore reale senza che il VMM intervenga.

3. **Controllo delle Risorse**: Il controllo delle risorse hardware deve essere di esclusiva competenza del VMM, senza interferenze da parte delle VM.

Generalmente il VMM ï¿½ composto da diversi moduli che gestiscono le macchine virtuali e le risorse della macchina reale.

Il primo modulo ï¿½ il *dispatcher*, il cui compito ï¿½ quello di richiamare gli altri moduli a seconda delle istruzioni eseguite dalla VM. Ad esempio, quando una macchina virtuale esegue un'istruzione che avrebbe l'effetto di modificare lo stato della macchina reale, riservando l'accesso a risorse hardware, il dispatcher intercetta l'istruzione e lascia il controllo al secondo modulo detto *allocator*.

Quest'ultimo si preoccupa di fornire alle macchine virtuali le risorse necessarie evitando i conflitti allo stesso modo di come farebbero le componenti di un sistema operativo nell'amministrazione dei processi. Esso ï¿½ senza dubbio la componente piï¿½ complessa del VMM.

Il terzo e ultimo modulo ï¿½ l'*interpreter*. Le macchine virtuali non possono avere accesso diretto alle risorse fisiche e non conoscono lo stato dell'hardware reale, ma solo quello del loro ambiente virtuale. Le istruzioni che fanno riferimento alle risorse devono essere simulate in modo da riflettere la loro esecuzione nell'ambiente delle macchine virtuali. Questo compito ï¿½ demandato all'interpreter.

2.2.3 Requisiti di Popek-Goldberg

La formalizzazione dei sistemi di calcolo alla quale si ï¿½ accennato nell'introduzione del paragrafo 2.2 assume che i processori supportino due stati d'utilizzo: *supervisor-mode* e *user-mode*.

Mentre in user-mode le istruzioni che compongono il set di istruzioni del processore (ISA) sono limitate al sottoinsieme di esse che non presenta problemi di sicurezza o che tipicamente non modificano lo stato dei registri, in supervisor-mode l'insieme di istruzioni eseguibili non ï¿½ limitato.

L'assunzione di un processore con due modalitï¿½ d'operazione ï¿½ valida anche per la quasi totalitï¿½ dei sistemi odierni.

Le istruzioni possono essere classificate in 3 gruppi:

1. **Privilegiate:** Le istruzioni privilegiate possono essere eseguite solo in supervisor-mode. Se eseguite in user-mode il processore causa un interrupt che viene generalmente gestito dal sistema operativo.

2. **Sensibili**: Sono le istruzioni che modificano lo stato delle risorse del sistema, oppure dipendono da esse.

3. **User**: Tutte le altre istruzioni che non sono nï¿½ privilegiate nï¿½ sensibili.

Questi gruppi non sono necessariamente disgiunti.

I requisiti di Popek-Goldberg dipendono dalla classificazione delle istruzioni di sopra e definiscono una condizione sufficiente alla virtualizzazione dei sistemi. Tale condizione puï¿½ essere riassunta come segue:

> *Per ogni calcolatore di terza generazione, la costruzione di un VMM ï¿½ sempre possibile se il set di istruzioni sensibili del calcolatore ï¿½ un sottoinsieme delle sue istruzioni privilegiate.*

In altre parole, un VMM puï¿½ essere costruito per ogni computer sul quale tutte le istruzioni che modificano o dipendono dallo stato della macchina reale causano un interrupt se eseguite all'interno di una VM. In questo modo ogni interrupt puï¿½ essere catturato dal VMM, il quale puï¿½ provvedere alla corretta esecuzione delle istruzioni in modo trasparente. Le istruzioni di tipo user, che costituiscono tipicamente la maggioranza delle istruzioni, sono invece direttamente eseguibili dal processore fisico. Questo consente ad un VMM di rispettare le caratteristiche di controllo delle risorse, efficenza ed equivalenza. Se esistessero istruzioni sensibili non privilegiate la loro esecuzione indiscriminata nelle VM interferirebbe con lo stato delle risorse fisiche e quindi con il corretto funzionamento del VMM e di tutte le altre VM.

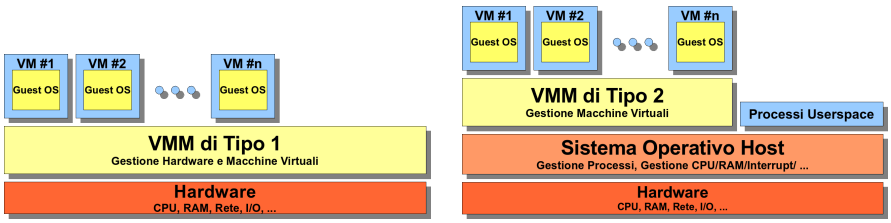
I requisiti di Popek-Goldberg, sebbene siano una condizione solo sufficiente a garantire la possibilitï¿½ di virtualizzazione di un'architettura, hanno notevoli conseguenze anche sulle architetture che non li soddisfano.

Si puï¿½ osservare che se un'architettura non rispetta i requisiti di Popek-Goldberg, deve contenere almeno un'istruzione sensibile ma non privilegiata. Quando una VM esegue un'istruzione sensibile non privilegiata, essa sfuggirï¿½ al controllo del VMM poichï¿½ l'istruzione non causerï¿½ un interrupt che possa essere intercettato e gestito. Sono quindi necessarie complesse tecniche che identifichino queste istruzioni prima della loro esecuzione. È ovvio che queste tecniche avranno il loro costo in termini di prestazioni e possono seriamente minare l'utilizzabilitï¿½ della virtualizzazione sulle architetture in questione.

Purtroppo la maggior parte delle architetture moderne utilizzate nei sistemi COTS non rispetta la condizione di Popek-Goldberg[3]. Tuttavia sono stati sviluppati sistemi di virtualizzazione anche per queste architetture adoperando tecniche che consentono di minimizzare la perdita di performance. Una di esse ï¿½ la *para-virtualizzazione* e verrï¿½ esposta piï¿½ avanti in questo capitolo.

[3]L'ISA dell'architettura x86 contiene diverse istruzioni sensibili non privilegiate.

2.2.4 Tipologie di VMM



(a) Rappresentazione schematizzata di un sistema di virtualizzazione che impiega un VMM di Tipo 1. Esso ï¿½ posto immediatamente sopra l'hardware.

(b) VMM di Tipo 2. In questo caso il VMM ï¿½ posto sopra il sistema operativo host.

Figura 2.1: Sistemi di virtualizzazione con VMM di tipo 1 e 2.

Esistono diversi sistemi di virtualizzazione sia commerciali che free-software e molti di essi derivano le proprie caratteristiche da ampi studi condotti da universitï¿½ e centri di ricerca. Molti di essi sono in grado di virtualizzare architetture che sono notoriamente problematiche poichï¿½ non possiedono i requisiti esposti sopra. In generale la componente piï¿½ importante di tutti questi sistemi ï¿½ rappresentata dal VMM. Essi si possono distinguere in due tipi:

Tipo 1: Il VMM ï¿½ posto immediatamente sopra l'hardware. Dal punto di vista della gestione della memoria, delle periferiche e del processore esso ï¿½ considerabile un sistema operativo o un kernel a tutti gli effetti. Rispetto ad essi, il VMM implementa in piï¿½ i meccanismi necessari per la gestione delle macchine virtuali. In 2.1 (a) si puï¿½ osservare la rappresentazione di un sistema di virtualizzazione che impiega un VMM di tipo 1. Le macchine virtuali al di sopra di esso possono eseguire un altro sistema operativo e vengono denominate macchine "guest".

Tipo 2: Il VMM ï¿½ un processo in esecuzione nell'ambiente del sistema operativo detto "host". È a quest'ultimo che sono demandate le funzioni di gestione della macchina fisica. Le macchine virtuali sono costituite da uno o piï¿½ sotto-processi del VMM. La figura 2.1 (b) mostra un sistema con un VMM di tipo 2.

2.3 Tipologie di Virtualizzazione

Da diversi anni esistono molteplici sistemi di virtualizzazione che consentono l'esecuzione di piï¿½ sistemi operativi completi sulla stessa macchina fisica. Essi sono in grado di offrire ai sistemi guest la condivisione di risorse quali connessioni di rete, accesso a periferiche di I/O e ovviamente CPU e memoria. Ognuno di essi adotta vari approcci, i piï¿½ comuni dei quali sono riassunti di seguito.

2.3.1 Virtualizzazione Completa

La virtualizzazione completa ï¿½ quel tipo di virtualizzazione che consente alle VM di eseguire sistemi operativi completi e *non modificati*. Ciï¿½ ï¿½ ottenuto tramite la completa simulazione in software di risorse hardware sulle quali le VM verrano eseguite.

I sistemi che implementano questo tipo di virtualizzazione consentono alle macchine virtuali di eseguire nativamente le istruzioni di tipo user, mentre le istruzioni sensibili vengono tradotte a run-time per mezzo di complessi meccanismi software detti di "binary translation".

Questi sistemi fanno tipicamente uso di VMM di tipo 2 e dipendono quindi dalla presenza di un sistema operativo host che permetta l'esecuzione del VMM il quale genera e gestisce le macchine virtuali.

L'efficenza di questo tipo di virtualizzazione ï¿½ in larga misura ridotta a causa della necessitï¿½ dei meccanismi di binary translation.

Questi meccanismi sono tuttavia necessari per ovviare alle limitazioni delle architetture che non possiedono i requisiti di Popek-Goldberg.

2.3.2 Para-virtualizzazione

La para-virtualizzazione nasce in risposta al degrado di performance dovuto alla binary translation necessaria nella virtualizzazione completa.

La para-virtualizzazione evita la traduzione esplicita delle istruzioni attraverso l'impiego di un VMM di tipo 1, che nel caso specifico viene denominato *hypervisor*, e che fornisce un'interfaccia software simile a quella di un sistema operativo.

I sistemi guest in esecuzione sulle macchine virtuali possono usufruire di tale interfaccia per avere un accesso filtrato alle risorse hardware che vengono comunque gestite a basso livello dall'hypervisor. L'interazione avviene per mezzo di "hypercalls" similari alle "supercall" che sfruttano i processi in un comune sistema operativo quando accedono alle risorse hardware attraverso il kernel.

Per poter utilizzare tale interfaccia, i sistemi operativi guest devono essere opportunamente modificati in modo da sostituire le componenti che interagiscono con l'hardware fisico nell'esecuzione non virtualizzata, con meccanismi che sfruttino l'interfaccia fornita dall'hypervisor.

2.3.3 Virtualizzazione a livello di SO

In questo tipo di virtualizzazione un singolo kernel fornisce funzioni che permettono l'esecuzione di pi¿½ ambienti indipendenti. Tuttavia le macchine virtuali di questo tipo devono essere immagini identiche del sistema operativo host. Inoltre l'isolamento delle macchine virtuali offerto da questo tipo di virtualizzazione i¿½ di grado inferiore rispetto alle altre poich¿½ ogni VM condivide lo stesso kernel. Una falla di sicurezza nel kernel si ripercuoterebbe quindi su tutte le VM.

2.3.4 Virtualizzazione Hardware-assisted

Dal 2005 i maggiori produttori di CPU di largo consumo, hanno introdotto nei propri processori delle estensioni che facilitano la gestione delle macchine virtuali e consentono di superare il degrado di performance in architetture che non rispettano i requisiti di Popek-Goldberg (come x86 e x86_64 implementate da AMD e Intel).

È possibile dotare, come del resto i¿½ gi¿½ stato fatto, un VMM della capacit¿½ di sfruttare queste estensioni consentendo una *virtualizzazione completa* ma con un degrado di prestazioni minimo, paragonabile o in alcuni casi superiore a quello offerto dalla para-virtualizzazione.

Ci¿½ i¿½ ottenuto attraverso la parziale implementazione in hardware dei complessi meccanismi quali la gestione delle istruzioni sensibili non privilegiate o la traduzione a run-time degli indirizzi di memoria delle macchine virtuali a indirizzi fisici, che normalmente deve gestire il VMM.

Tuttavia, questo tipo di virtualizzazione i¿½ strettamente dipendente dalle estensioni hardware di cui si i¿½ parlato, e quindi l'utilizzo i¿½ riservato ad architetture che le implementano. Non va per¿½ tralasciato il fatto che la diffusione di queste CPU i¿½ destinata a crescere notevolmente.

2.4 Il Modello di Xen

Xen i¿½ un sistema di para-virtualizzazione free software a sorgente aperto per le architetture x86, x86_64, IA-64 e PowerPC anche in configurazione

SMP. In questo paragrafo si vedranno le caratteristiche principali, insieme ad una panoramica dell'architettura di Xen.

2.4.1 Introduzione

Xen nasce come progetto di ricerca dell'universitï¿½ di Cambridge e a partire dal 2007 ï¿½ diventato un prodotto rivolto anche al mercato enterprise. La prima versione fu rilasciata nel 2003. Esso ï¿½ stato realizzato principalmente per permettere l'esecuzione contemporanea di piï¿½ sistemi operativi *guest* sullo stesso computer.

Xen introduce una terminologia differente per distinguere il sistema operativo host, che viene detto *Domain 0*, spesso abbreviato con *Dom0*, da quelli guest denominati *Domain U* (*DomU*). Il Dom0 gode di un accesso privilegiato all'hardware e collabora con l'Hypervisor per l'amministrazione delle risorse fisiche. Per questa caratteristica il Dom0 viene anche indicato con "*Privileged Domain*"

Come Dom0 si possono utilizzare i sistemi operativi Linux, NetBSD o Solaris, tuttavia questi sistemi devono essere opportunamente modificati.

La compatibilitï¿½ per i sistemi guest ï¿½ molto aumentata a partire dalla fine del 2006 da quando Xen supporta le estensioni per la virtualizzazione delle architetture sopra indicate, che consentono una virtualizzazione hardware-assisted e l'impiego di sistemi operativi guest non modificati.

Infatti, accanto a sistemi "para-virtualizzati" (Linux, xBSD, Solaris, Plan 9, ecc.) che necessitano di modifiche al kernel, possono essere impiegati come DomU sistemi non modificabili come ad esempio il sistema proprietario MS Windows.

Alla base del sistema di Xen vi ï¿½ lo *Xen hypervisor*, ovvero un VMM costituito da un "sottile" strato software in esecuzione immediatamente sopra l'hardware. Attraverso di esso Xen si propone di risolvere gli svantaggi insiti nell'impiego della virtualizzazione completa offrendo ai sistemi operativi guest un'astrazione software simile all'hardware sottostante.

2.4.2 Architettura e Caratteristiche di Xen

Come ï¿½ giï¿½ stato detto un ambiente Xen consta di vari oggetti che compartecipano alla creazione dell'ambiente virtualizzato. La descrizione che segue ï¿½ basata sulla versione 3.0 di Xen per architettura x86 con sistema operativo GNU/Linux.

Hypervisor

L'hypervisor di Xen ï¿½ responsabile per lo scheduling della CPU e il partizionamento della memoria tra le macchine virtuali in esecuzione su di esso.

Oltre ad astrarre le risorse hardware, esso controlla l'esecuzione delle macchine virtuali. Cïï¿½ avviene in modo del tutto simile allo scheduling operato dai comuni sistemi operativi per i processi in esecuzione. Ovvero le macchine virtuali vengono sospese e riavviate trasparentemente e non hanno accesso diretto all'hardware ma possono soltanto invocare i servizi offerti dall'hypervisor.

L'hypervisor mantiene traccia dello stato delle risorse e ne regola l'assegnazione sulla base di politiche che ne ottimizzino l'utilizzo.

L'analogia con un sistema operativo non puï¿½ procedere oltre poichï¿½ l'hypervisor non ha alcuna conoscenza delle periferiche di rete, storage, video o qualunque altro tipo di I/O.

Al momento dell'accensione della macchina l'hypervisor viene caricato dal boot loader e, dopo aver riservato una minima porzione di memoria per se ed inizializzato le strutture dati per la gestione delle componenti da virtualizzare, lascia il controllo al dominio privilegiato.

Domain 0

Il *Domain 0* ï¿½ nel nostro caso un sistema operativo basato su una versione modificata del kernel Linux. Esso, sebbene non sia strutturalmente differente dalle altre macchine virtuali eventualmente in esecuzione sull'hypervisor, riveste un compito molto importante.

Infatti ï¿½ attraverso il Dom0 che avviene l'accesso alle periferiche non direttamente gestite dall'hypervisor e perciï¿½ esso gode di un accesso privilegiato alle risorse hardware. Questo consente ai sistemi Xen di poter sfruttare l'ampia selezione di *device drivers* sviluppati per il kernel di Linux che negli ultimi anni ha conosciuto un espansione notevole. Il Dom0, interagendo con le macchine DomU, offre a quest'ultime l'accesso alle periferiche non gestite dall'hypervisor, consentendo una notevole semplificazione del codice di quest'ultimo.

Il Dom0 fornisce un interfaccia di controllo per le funzioni dell'hypervisor e consente di separare i meccanismi (implementati nell'hypervisor) dalle politiche di gestione (gestite appunto nel Dom0). Inoltre ï¿½ attraverso il Dom0 che l'amministratore di sistema puï¿½ creare, gestire e distruggere le istanze di DomU. È quindi necessario che il Dom0 sia il primo ambiente ad essere eseguito dall'hypervisor.

Nella fase di boot infatti, dopo che il controllo ï¿½ stato lasciato dal boot loader all'hypervisor, quest'ultimo provvede a riservare la memoria sufficiente al kernel del Dom0, che viene caricato, immediatamente aggiunto alla lista delle macchine virtuali ed avviato. In seguito esso, seguendo la normale procedura d'avvio di un sistema Linux, provvederï¿½ al caricamento dei driver necessari.

I Dom0, oltre ai comuni driver, includono due particolari "backend" utilizzati per virtualizzare l'accesso da parte dei DomU alle periferiche di rete e ai dispositivi a blocchi. Essi sono rispettivamente il "*Network Backend Driver*" e il "*Block Backend Driver*". Attraverso tali strumenti il Dom0 ï¿½ in grado di accogliere le richieste dei DomU permettendo ad essi l'accesso alla rete e allo storage.

Domain U

Tutte le macchine para-virtualizzate o virtualizzate con l'impiego di estensioni hardware, vengono denominate *Domain U* ovvero *Unprivileged Domains*.

I sistemi operativi dei DomU effettuano le loro normali operazioni di scheduling dei processi e gestione della memoria sulle risorse virtualizzate dall'hypervisor, sgravando quest'ultimo da tali compiti. A differenza di altri tipi di virtualizzazione le modifiche apportate al kernel dei DomU fa si che essi siano a conoscenza del loro stato di "macchina virtuale" e agiscano di conseguenza. Infatti tutte le operazioni di accesso all'hardware vengono sostituite da hypercalls, ovvero chiamate all'interfaccia esportata dall'hypervisor. Attraverso di esse i Dom0 possono richiedere ad esempio di riservare pagine di memoria per i propri processi.

L'accesso alle periferiche avviene per mezzo dei driver di "frontend" di rete e block devices, che rappresentano le controparti nel dialogo con i backend ospitati dal Dom0. La comune gestione delle periferiche attraverso interrupt operata dai sistemi operativi con accesso diretto all'hardware, viene sostituita da un meccanismo ad eventi che vengono segnalati al corretto DomU.

In questo modo l'hypervisor puï¿½ catturare gli interrupt provenienti dalle periferiche hardware e generare gli eventi di cui un determinato DomU ï¿½ in attesa, come ad esempio la presenza di dati da poter leggere su un socket.

Le immagini dei sistemi operativi guest possono essere ospitate su un'ampia varietï¿½ di supporti visti dai domU come *dispositivi a blocchi virtuali* (VBD)[4]. Essi possono essere ricavati da:

- Dispositivi Fisici: partizioni fisiche ospitate da un dispositivo a blocchi.

[4]Virtual Block Devices

```
name = "DomU1"
#quantitï¿½ di memoria da assegnare a DomU1
memory = 1024
#numero di CPU virtuali
vcpu = 2
#immagine del kernel modificato
kernel = "/boot/vmlinuz-2.6.16-1-xen"
#ip da assegnare all'interfaccia di rete virtuale
vif = [ 'ip=192.168.1.10' ]
#dispositivi a blocchi virtuali
disk = [ 'phy:/dev/iscsi_02,hda2,w','phy:/dev/iscsi_01,hda1,w' ]
#dispositivo virtuale contenente il root file system
root = "/dev/hda2 ro"
```

Tabella 2.1: File di configurazione per la definizione di un DomU.

- File: immagini di file system su file.
- LVM: volumi logici.
- NFS: file system forniti come export NFS[5]

I domU sono definiti attraverso comuni file di testo che, attraverso un preciso formato, permettono di specificare le caratteristiche della macchina da creare. Un esempio ï¿½ fornito in tabella 2.1:

Demoni e Strumenti di Gestione

La gestione dell'ambiente virtuale viene effettuata attraverso una serie di demoni[6] e programmi che devono essere eseguiti sul Domain 0 e che permettono l'interazione con l'hypervisor. La figura 2.2 mostra lo schema di tali componenti. Tra di essi il principale ï¿½ il demone *xend*.

Xend ha il compito di fungere da tramite tra l'amministratore del sistema e l'hypervisor che ne gestisce a piï¿½ basso livello l'allocazione delle risorse. Esso fa uso della libreria *libxenctrl* per dialogare con lo speciale driver *privcmd* che rappresenta l'interfaccia di comunicazione attraverso il quale il Dom0 presenta all'hypervisor le richieste sottomesse dall'amministratore.

[5]Network File System. Consente di esportare directory ospitate fisicamente da una macchina attraverso la rete.

[6]In ambienti UNIX col termine *demone (deamon)* ci si riferisce ai processi di sistema in esecuzione non interattiva che offrono servizi e funzioni di gestione

Le richieste da sottomettere all'hypervisor vengono gestite attraverso l'applicazione a riga di comando *xm* che offre agli amministratori un interfaccia che consente di creare, spegnere, riavviare, distruggere e monitorare le macchine virtuali.

Ad esempio, nella creazione di un domU, l'amministratore specifica i parametri della VM da creare visti nel precedente paragrafo e attraverso l'invocazione del comando "*xm create*" xend comunica all'hypervisor le relative richieste di allocazione delle risorse. In seguito xend provvederï¿½ anche al caricamento del sistema operativo guest.

Accanto Xend vi ï¿½ il demone *xenstored* che gestisce parte dello scambio di informazioni tra il Dom0 e i DomU e mantiene delle strutture dati necessarie per effettuare lo scambio dati da e verso i dispositivi di rete e a blocchi per conto dei DomU.

La gestione dei dispositivi infatti avviene con la mediazione di quest'ultimo demone, responsabile della corretta gestione dei canali di comunicazione, detti *event channel*, atti a sostituire i comuni meccanismi di interrupt, che non vengono piï¿½ gestiti dai singoli sistemi operativi guest, bensï¿½ catturati dall'hypervisor il quale si incarica di notificare per mezzo di un sistema ad eventi gli interrupt provenienti dai dispositivi di I/O. I sistemi operativi guest reagiscono agli eventi in maniera analoga a quanto avviene con gli interrupt nei sistemi operativi non modificati.

Lo scambio di dati vero e proprio da e verso i dispositivi avviene in modo asincrono e per mezzo di aree di memoria condivisa sotto l'arbitrio dell'hypervisor che assicura l'isolamento degli spazi di memoria delle VM.

2.4.3 Performance

Non esistono ancora modalitï¿½ di benchmark[7] universalmente accettate per le prestazioni dei sistemi di virtualizzazione. Tuttavia si puï¿½ affermare che essi devono essere valutati in riferimento ad almeno due aspetti fondamentali:

- Perchï¿½ siano valide le proprietï¿½ di efficenza ed equivalenza esposte nei paragrafi precedenti, ï¿½ necessario che l'impatto prestazionale negativo su un programma in esecuzione in un ambiente virtualizzato sia minimo. È auspicabile quindi che, per un programma di test, la differenza del tempo d'esecuzione in un sistema operativo guest e sul medesimo sistema eseguito sulla macchina fisica, sia minima.

- In secondo luogo, ï¿½ importante che il degrado di performance aumenti con andamento lineare dipendentemente dal numero delle istanze di

[7] Il termine *benchmark* ï¿½ usato per indicare un insieme di applicazioni di test volte a valutare le prestazioni dei sistemi di calcolo in determinati ambiti.

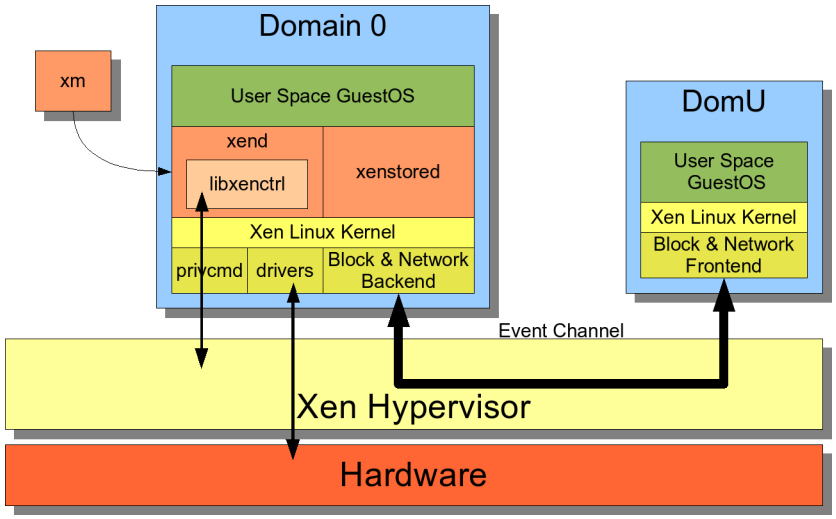


Figura 2.2: Schema riassuntivo dell'architettura di Xen.

macchine virtuali in esecuzione sulla stessa macchina host. Ciï¿½ ï¿½ sinonimo di un'equa distribuzione delle risorse tra le macchine virtuali e un overhead dovuto alla gestione delle macchine praticamente trascurabile.

In figura 2.3 sono illustrati alcuni dei risultati ottenuti attraverso il calcolo della mediana dei valori riportati da varie esecuzioni consecutive di diversi benchmark. I test riportati, effettuati dal gruppo di ricerca di Xen, sono volti a mostrare le differenze di prestazioni tra un ambiente GNU/Linux non virtualizzato e uno virtualizzato.

Le barre contrassegnate con "*L*" mostrano il risultato del sistema operativo (Linux) sulla macchina fisica, mentre quelle contrassegnate con "*X*" si riferiscono al risultato dei medesimi benchmark eseguiti su una VM Xen.

Le prime due barre mostrano il confronto dei risultati del benchmark SPEC CPU, ovvero una serie di test *CPU-Bound* in userspace, con brevi operazioni di I/O e alto tasso di utilizzo della CPU e della RAM. Il valore riportato ï¿½ il punteggio assegnato dal benchmark.

La seconda coppia di barre mostra il tempo impiegato in secondi per la compilazione del kernel Linux. Questa operazione utilizza intensivamente la CPU ed esegue molte operazioni di lettura e scrittura da e su file.

Le successive due coppie riportano il risultato dei benchmark OSDB-OLTP e OSDB-IR[8], specifici per la valutazione delle performance di database,

[8]Open Source Database Benchmark - OnLine Transaction Processing e Open Source Database Benchmark - Information Retrieval

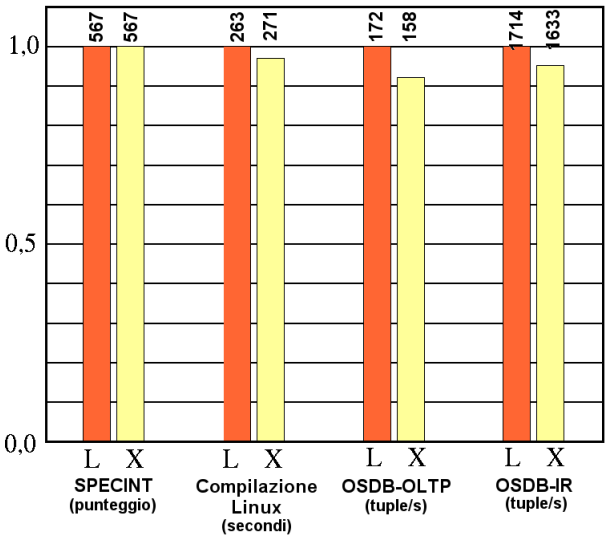


Figura 2.3: Performance relative di Xen rispetto ad un ambiente GNU/Linux non virtualizzato. I dati sono tratti dall'articolo *Xen and the Art of Virtualization [1].*

ovvero operazioni tipicamente orientate all'I/O. Il valore mostrato ï¿½ in tuple accedute o modificate al secondo.

Alla luce dei risultati di benchmark, si puï¿½ asserire che la differenza di performance nel caso nativo e virtualizzato ï¿½ trascurabile per applicazioni con alto tasso d'utilizzo della CPU. Mentre le applicazioni che compiono frequenti operazioni di I/O sono solo lievemente penalizzate dall'impiego in un ambiente virtualizzato. In generale, il degrado prestazionale introdotto dallo Xen hypervisor ï¿½ inferiore al 5%.

Per quanto riguarda l'aspetto del degrado di performance con crescita lineare, Xen ï¿½ stato appositamente progettato per offrire un'elevata scalabilitï¿½, con l'ambizioso obiettivo di eseguire concorrentemente fino ad oltre 100 macchine virtuali sulla stessa macchina host. Sono molteplici i test effettuati anche in questo ambito. Essi basano la propria analisi sulla crescita lineare del tempo di calcolo nell'esecuzione di programmi di test, all'aumentare delle VM in esecuzione. Xen mostra una crescita in molti casi coincidente al caso ideale, ovvero esattamente lineare [1] [2] [3] [4] [5].

Le valutazioni citate mettono in mostra la validitï¿½ dell'approccio alla virtualizzazione delle tecniche di para-virtualizzazione ed in particolare di quelle impiegate da Xen. A fronte dei vantaggi offerti, le contenute perdite di performance candidano Xen a poter essere impiegato in ambienti dove le alte prestazioni giocano un ruolo importante, come i cluster e le griglie computazionali.

2.5 Applicazioni della Virtualizzazione

I vantaggi offerti dalla virtualizzazione sono molteplici. Di seguito vengono mostrate le ragioni del riacceso interesse verso queste tecniche.

Maggiore sfruttamento dell'hardware (consolidamento): Molto spesso i servizi in esecuzione su una determinata macchina non sono sufficienti a saturare la capacitï¿½ delle risorse che essa possiede.

Attraverso l'esecuzione di piï¿½ VM, il numero di servizi offerti puï¿½ crescere fino a ridurre l'inutilizzo delle risorse.

Isolamento: Si puï¿½ sfruttare l'esecuzione di piï¿½ VM per isolare ambienti sicuri, da altri potenzialmente a rischio. Ad esempio, servizi poco affidabili e da esporre al pubblico possono essere implementati su VM, senza compromettere la sicurezza dell'intero sistema.

Controllo delle risorse: Ad ogni macchina virtuale si possono assegnare risorse in base a precisi criteri. Ciï¿½ evita che alcune applicazioni possano sfruttare eccessivamente CPU, memoria od altro a discapito di altre applicazioni che vedrebbero la propria funzionalitï¿½ ridotta.

Test: Attraverso la virtualizzazione si puï¿½ disporre di ambienti di complessitï¿½ variabile, anche distribuiti, senza la necessitï¿½ di dotarsi di piï¿½ hardware. Ciï¿½ consente il test di molte applicazioni che necessitano di una tale complessitï¿½ d'ambienti, come ad esempio applicazioni parallele o peer-to-peer. Inoltre le applicazioni possono essere testate per piï¿½ sistemi operativi che possono essere ospitati dalle VM.

Facilitï¿½ d'amministrazione: L'implementazione di una macchina virtuale richiede molto meno sforzi da parte degli amministratori. Inoltre le macchine virtuali possono essere fermate, messe in pausa, riavviate o addirittura"clonate" su altre macchine fisiche senza dover interrompere il funzionamento della macchina reale. Ciï¿½ consente di utilizzare la virtualizzazione per implementare meccanismi di fail-over o di bilanciamento del traffico molto piï¿½ efficaci rispetto alla semplice replicazione hardware.

Capitolo 3

Prototipo: Analisi

3.1 Introduzione

Nei precedenti capitoli si è visto che l'insorgere di molti dei problemi legati alle infrastrutture di calcolo di natura condivisa è dovuto alle crescenti necessità eterogenee che aumentano considerevolmente in dipendenza dal numero di entità che mettono in condivisione le proprie risorse. Nel capitolo che segue si vogliono analizzare le problematiche inerenti questo aspetto, e si presenteranno le motivazioni alla base del prototipo oggetto di questa tesi.

Verranno inoltre presentate le linee guida e le scelte architetturali sulle quali il prototipo è stato sviluppato.

3.2 Problematiche Affrontate

In cluster medio-grandi come il cluster INFN della sezione di Perugia, il problema dell'eterogeneità raggiunge dimensioni notevoli e viene ulteriormente amplificato nelle realtà emergenti delle griglie computazionali. Esse devono infatti essere in grado di aggregare un gran numero di risorse eterogenee, concertandone l'utilizzo per offrire molte tipologie di servizi.

Di seguito sono esposte le principali problematiche cui il prototipo sviluppato vuole rispondere.

3.2.1 Impiego Sub-ottimale delle Risorse

Durante la crescita in dimensioni e nell'offerta di funzionalità dei cluster, si assiste ad un aumento del numero delle macchine che implementino i servizi richiesti. Ci si può rendere facilmente conto che tale numero cresce rapidamente se si considera che alcuni dei compiti che le macchine sono chiamate

a svolgere necessitano di un alto grado di isolamento; ovvero abbisognano di ambienti d'esecuzione che presentano delle incompatibilità tra loro e che le costringono ad un utilizzo esclusivo.

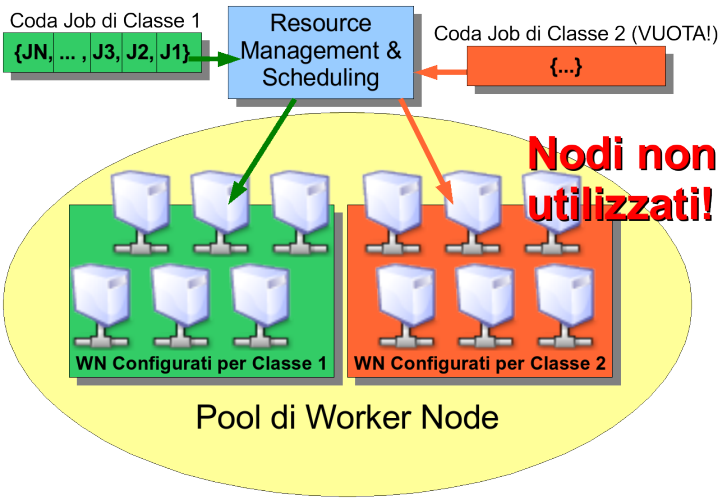


Figura 3.1: Esempio di uso sub-ottimale a causa del partizionamento del cluster: I nodi sono suddivisi in due classi per accogliere altrettante tipologie di job tra loro incompatibili. La componente di RMS raccoglie le richieste tramite due code separate e distribuisce il carico in accordo alla distinzione in classi. In periodi di scarso tasso di sottomissione di job di classe 2 i nodi adibiti all'esecuzione di quest'ultimi non possono essere impiegati per l'altra classe di job restando di fatto inattivi, mentre potrebbero essere sfruttati per i job di classe 1 in coda.

Alcune caratteristiche degli ambienti d'esecuzione infatti, sono spesso difficili da conciliare o a volte del tutto contrastanti: diverse versioni di librerie, compilatori, interpreti, programmi, servizi di sistema, ecc.

Spesso gli amministratori dei cluster sono costretti a fronteggiare l'impossibilità di fornire un unico ambiente d'esecuzione da installare su tutti i nodi, e passare all'inevitabile partizionamento delle macchine in classi, dedicate al soddisfacimento delle richieste di servizi che necessitano di ambienti particolari.

È ad esempio il caso della gestione delle sottomissioni di job che richiedono ambienti d'esecuzione specifici e tra loro incompatibili. Da questo punto di vista i job possono essere divisi in classi sulla base delle diverse necessità, divisione che si riflette sul partizionamento delle risorse. Ancor più specificatamente, nel cluster INFN di Perugia si è visto che vi è una divisione tra i nodi impiegati per l'utenza INFN-Grid e quella locale. A loro volta, entrambe le classi d'utenza presentano delle ulteriori divisioni in base al gruppi di ricerca afferenti a diversi esperimenti.

Questo approccio porta ad un utilizzo delle singole macchine fisiche ben al di sotto del loro limite, conducendo ad un impiego sub-ottimale delle risorse di calcolo. Infatti si possono presentare condizioni nelle quali la distribuzione delle sottomissioni sia molto sbilanciata verso una determinata classe di job, utilizzando intensamente le macchine appartenenti alla classe corrispondente e lasciando in periodi di sostanziale inutilizzo quelle appartenenti ad altre. In figura 3.1 è mostrata una simile circostanza.

La migrazione delle risorse da una classe ad un'altra è spesso non praticabile o comunque ardua da attuare in tempi brevi tali da rispondere ad evenienza del genere, poiché presuppone la riconfigurazione delle macchine da parte degli amministratori, con i disagi che ne conseguono (tempi di downtime, impiego di risorse umane, difficoltà d'amministrazione, ecc.). Il cluster è quindi limitato nell'adattarsi alle variazioni nel tempo della tipologia di richieste.

Va inoltre aggiunto che le classi di servizi incompatibili possono essere molte, e i dettagli che differenziano le une dalle altre molto difficili da gestire e razionalizzare. Ne consegue che, oltre ad aumentare il frazionamento della destinazione d'uso delle risorse, cresca notevolmente la difficoltà di gestione del cluster.

3.2.2 Gestione Poco Flessibile delle VM

Le tecniche di virtualizzazione sono ormai da tempo efficacemente adoperate per aumentare il tasso di utilizzo delle macchine fisiche e offrire vantaggi in termini di isolamento, facilità d'amministrazione e disponibilità, superando in parte il problema dell'utilizzo inefficiente.

Tuttavia la gestione delle macchine virtuali è generalmente piuttosto statica, ovvero le macchine virtuali devono essere configurate e lanciate secondo necessità dagli amministratori dei sistemi. Ciò limita la flessibilità offerta dalla virtualizzazione poiché le macchine così gestite non sono in grado di rispondere ai bisogni nei tempi molto brevi spesso richiesti.

L'uso di macchine virtuali resta un ottimo sistema per il consolidamento dell'offerta dei servizi stabili nel tempo, ma in realtà, sono molte le macchine che vengono adoperate per svolgere compiti in periodi ben definiti. È il caso delle workstation impiegate nei laboratori didattici, e di tutti i server dedicati all'offerta di servizi interattivi per gli utenti, che tipicamente hanno degli orari d'utilizzo giornalieri stabiliti da regole d'accesso.

Inoltre, molto spesso studenti, ricercatori e insegnanti hanno la necessità di adoperare risorse con particolari configurazioni per il test di applicazioni in fase di sviluppo, o esercitazioni di vario genere.

In questi casi la configurazione manuale di macchine virtuali per ogni singolo computer, in modo da soddisfare le richieste o impiegarli come nodi di calcolo nei tempi di inutilizzo, risulta alquanto difficoltosa.

3.3 Ambienti Virtuali

In questo paragrafo si vuole presentare l'astrazione principale adoperata dal prototipo per fornire una soluzione alle problematiche esposte sopra. Le difficoltà di cui si è parlato dipendono principalmente dall'impossibilità di offrire all'interno dello stesso ambiente d'esecuzione la varietà di servizi richiesti. Gli *ambienti virtuali*, di cui a breve si darà una definizione più chiara, costituiscono il punto di partenza per superare questi ostacoli.

3.3.1 Definizione

Un ambiente d'esecuzione è tipicamente costituito da un sistema operativo e da una collezione di librerie, programmi e servizi di sistema volti all'espletamento di particolari funzioni richieste dall'utenza. Generalmente tutte queste componenti devono essere installate dagli amministratori, andando incontro alle problematiche osservate in precedenza, dovute a richieste mutualmente incompatibili.

Gli ambienti d'esecuzione possono essere in parte astratti dalle macchine fisiche che li eseguono, con il limite non banale della compatibilità con l'hardware. Ciò consente di configurare le macchine in modo che possano eseguire in tempi diversi ambienti d'esecuzione differenti.

È infatti possibile dotare i computer di diverse immagini di sistemi operativi configurati secondo le caratteristiche di un particolare ambiente d'esecuzione e selezionare quello richiesto in fase di boot. Tuttavia ciò comporta svantaggi quali il necessario riavvio della macchina e l'installazione degli ambienti su ogni macchina.

Le peculiarità della virtualizzazione ben si coniugano con questo aspetto, permettendo la definizione di *ambienti d'esecuzione virtuali*[1] che, rispetto a quelli canonici, godono di un grado di astrazione dalle macchine fisiche ben più alto.

Ciò è ottenuto attraverso la definizione di macchine virtuali preposte a fornire determinati ambienti d'esecuzione.

Un ambiente virtuale oltre a poter essere modellato nelle sue caratteristiche software in risposta alle esigenze, consente una quasi completa in-

[1] Da ora in avanti *ambienti virtuali*. In alcuni casi ci si riferirà ad essi con *domini dinamici*, mutuando la terminologia di Xen.

dipendenza dallo strato hardware, permettendo di definire entro certi limiti anche le caratteristiche fisiche delle macchine virtuali stesse (numero di CPU virtuali, quantità di RAM, spazio disco, capacità di rete, ecc).

Accanto a ciò si può beneficiare di tutte le caratteristiche che i sistemi di virtualizzazione offrono: maggiore sfruttamento dell'hardware attraverso l'esecuzione di più ambienti virtuali su una singola macchina fisica, maggiore sicurezza e isolamento, facilità d'amministrazione, ecc.

Un ambiente virtuale è quindi definibile attraverso due insiemi di caratteristiche:

- Configurazione del software dell'immagine del sistema operativo e quindi dell'ambiente d'esecuzione.
- Caratteristiche dell'hardware virtuale.

L'approccio impiegato per la realizzazione degli ambienti virtuali è il seguente: la configurazione della dotazione software avviene per mezzo dell'installazione del sistema operativo e dei pacchetti software necessari in modo da poter ottenere un file-immagine. Insieme ad esse, i file di configurazione di Xen[2] consentono di specificare tutte le altre caratteristiche necessarie alla definizione di un ambiente virtuale.

Ogni ambiente virtuale può essere in questo modo facilmente replicato su altre macchine attraverso la semplice copia dell'immagine e del file di configurazione. Le immagini possono poi essere eventualmente replicate su supporti di altro tipo come partizioni o volumi logici.

Ciò consente di definire una volta per tutte un determinato ambiente evitando l'installazione manuale su ogni macchina virtuale. L'impiego naturale è quello di definire un ambiente virtuale per ogni classe di servizi incompatibili e replicarli quando richiesto su macchine fisiche dotate di un sistema di virtualizzazione (nel nostro caso Xen).

3.3.2 Il Ruolo del Prototipo

Adoperando la terminologia di Xen, un sistema che gestisca automaticamente un pool di Domain 0, e su di essi attivi una serie di ambienti virtuali (Domain U), in risposta alle richieste e le disponibilità contingenti, permetterebbe di migliorare notevolmente i tempi di soddisfacimento delle richieste di servizi, eliminando la necessità di interventi umani.

Inoltre, un sistema di questo tipo permetterebbe l'adattamento dinamico delle risorse impiegandole per specifici compiti e diminuendone l'utilizzo

[2] Un esempio è stato fornito in 2.4.2

inefficiente. Ad esempio, richiamando il paragrafo precedente, si potrebbero definire tanti ambienti virtuali quante sono le classi di job incompatibili ed, attraverso l'interazione con una componente di management, gestirne in base alle richieste e alle disponibilità, la creazione e la distruzione.

Il prototipo che verrà esposto nel prosieguo di questo e del prossimo capitolo mira alla realizzazione di uno strumento in grado di reagire alle richieste e ad adattarsi in tempi brevi alle necessità, con lo scopo di ottimizzare l'utilizzo delle risorse e il grado di soddisfacimento delle richieste.

Nel paragrafo 3.7 si daranno ulteriori dettagli dell'interazione del prototipo con un batch system.

3.4 Architettura del Prototipo

In questo paragrafo verrano presentate le caratteristiche salienti del prototipo. L'implementazione segue un approccio modulare per consentire un'alta possibilità di configurazione e di adattamento a vari compiti.

3.4.1 Schema Generale del Prototipo

In figura 3.2 è mostrato lo schema ad alto livello del prototipo per la gestione dinamica degli ambienti virtuali. L'applicazione ha una struttura client server similare a quella di una componente di resource scheduling e management. La relazione tra il server, detto *Manager* e i client è di tipo master/slave, ovvero il Manager occupa un livello gerarchico logicamente superiore rispetto ai client, i quali sono subordinati alle sue richieste.

Il Manager è dunque la componente centrale del sistema. Esso è infatti preposto allo svolgimento dei seguenti compiti:

- Interrogazione dei client.
- Concentramento delle informazioni.
- Logica di decisione.
- Invio di comandi da eseguire sui client.

I client sono in esecuzione sulle macchine da monitorare e sui Dom0 sui quali si ha necessità del controllo da parte del Manager. Ogni client è in grado di accedere alle informazioni sensibili riguardanti l'ambiente nel quale è in esecuzione e di fornirle al Manager quando richiesto. Inoltre essi possono eseguire istruzioni sulla base delle decisioni del manager.

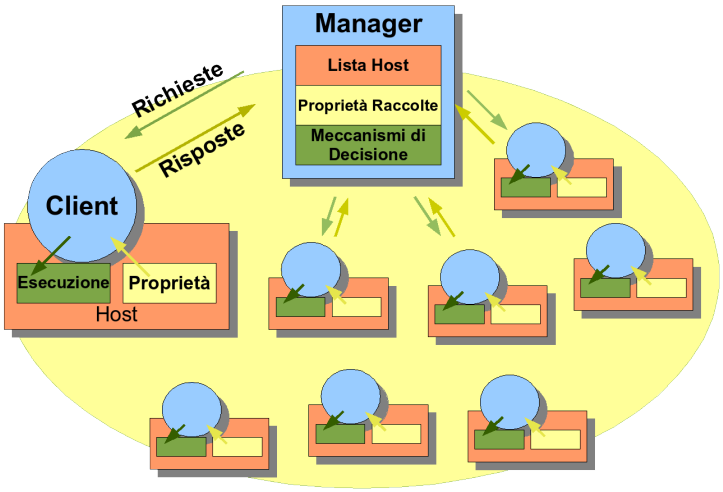


Figura 3.2: Schema generale del prototipo per la gestione di ambienti virtuali dinamici.

La struttura modulare dei client consente di adoperarli per più compiti su più tipologie di macchine. Le funzioni che i client devono tipicamente implementare sono:

- Raccolta di informazioni sensibili (*Proprietà*).
- Esecuzione di comandi per conto del Manager.

Di seguito sono presentate in dettaglio le componenti del manager, dei client e del canale di comunicazione tra di essi.

3.4.2 Manager

Il manager è un programma demone in esecuzione su una macchina all'interno della rete e svolge compiti di resource management. Esso mantiene una lista di tutti i client presenti. Periodicamente, il manager interroga i client richiedendo le informazioni che essi esportano. In seguito alla ricezione, queste informazioni, indicate con il termine *proprietà*, sono mantenute in apposite strutture dati.

Le informazioni raccolte sono divisibili in due tipologie: *stato delle risorse d'interesse* e *necessità del sistema*. Sulla base di queste informazioni il manager può decidere l'esecuzione di una serie di comandi sui client, in risposta al verificarsi di determinati eventi.

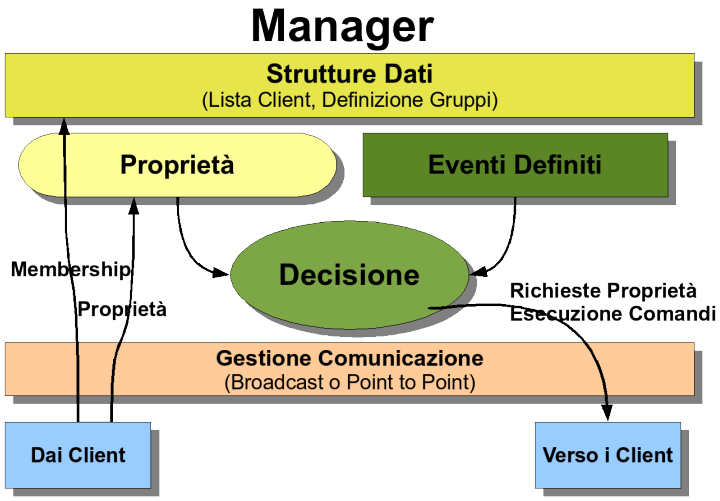


Figura 3.3: Schema della struttura interna del Manager.

La definizione degli eventi e dei comandi da eseguire in risposta ad essi avviene attraverso la configurazione del manager.

L'insieme di eventi costituisce la logica di decisione del manager e condiziona il comportamento di tutto il sistema. Più avanti, in questo capitolo saranno forniti maggiori dettagli su questo aspetto.

3.4.3 Client

Anche i client si comportano come demoni e come si è già accennato possono essere in esecuzione sia sui Dom0 che su altre macchine.

Fino ad ora non si è specificato quali sono le informazioni che i client esportano nè i comandi che essi possono eseguire in base alla decisione del manager. La flessibilità di configurazione introdotta, consente di definire per ogni client una lista di *proprietà*. Una proprietà è definita in base ad un'associazione di un'etichetta a un determinato valore di interesse. Ad esempio ogni client può definire la proprietà "memfree" indicando con essa la quantità di memoria RAM non utilizzata.

Le proprietà non hanno limitazioni riguardo al tipo, esse infatti devono poter contenere valori interi, stringhe, valori di verità, e così via.

I meccanismi con cui le informazioni vengono raccolte sono demandate interamente ai client e possono essere specificati attraverso script o programmi che ritornano i valori richiesti.

Ogni client aggiorna periodicamente in modo automatico i valori delle proprietà e le mantiene in attesa della richiesta del Manager.

Similmente alle proprietà, per ogni client si possono definire una serie di comandi accessibili dal Manager. Ad esempio si possono definire comandi batch che sincronizzano un'immagine di una VM sull'hard-disk locale alla macchina dove il client è in esecuzione.

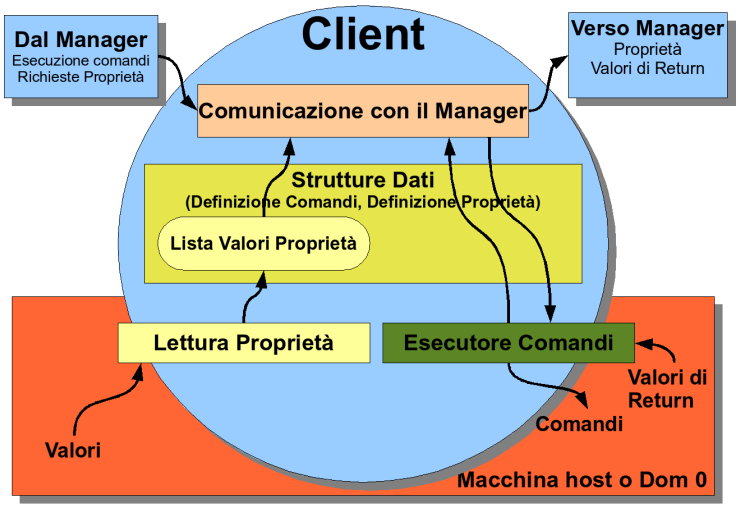


Figura 3.4: Schema della struttura interna della componente client.

Comandi e proprietà possono essere definiti in modo del tutto indipendente da un client all'altro in modo da poter tenere in considerazione le caratteristiche particolari di ciascun ambiente.

I client possono essere associati a gruppi per meglio gestire le macchine appartenenti ad una certa classe. Ciò consente ad esempio di definire un gruppo che contenga tutti i nodi di un cluster equipaggiati con 4 CPU fisiche, o appartenenti ad un determinato gruppo di ricerca, o classificabili in base ad altre caratteristiche.

3.5 Comunicazione

Lo scambio di informazioni e comandi tra client e manager avviene attraverso un semplice protocollo di comunicazione che supporta l'invio di messaggi in broadcast a sotto gruppi di client. I messaggi vengono gestiti in modo asincrono da manager e client. L'asincronia consente di ridurre il numero

di scambi di messaggi necessari tra manager e client; in più il supporto al broadcast consente al manager di gestire in modo naturale la comunicazione con intere classi di client.

I messaggi sono distinti in base ad un codice che stabilisce la semantica del contenuto informativo.

Inoltre il sistema di comunicazione scelto consente di definire gruppi sulla rete e di stabilire l'appartenenza di un client ad uno di essi tramite messaggi di "membership". Il manager può quindi sapere quanti sono i client connessi e a che classi appartengono.

3.6 Decisione

La struttura presentata consente di adattare il sistema a varie applicazioni di monitoring e controllo e l'architettura modulare consente di separare le politiche di gestione dai meccanismi. Si è detto che il manager basa la propria logica di decisione sulla definizione di eventi. Un generico evento può essere definito a partire da una semplice espressione booleana relativa alle proprietà che il manager riceve dai client o su altre informazioni che esso stesso possiede.

Evento 1:	Sono passati 5 minuti dall'ultimo aggiornamento delle proprietà.
Azione 1:	Richiedi invio proprietà a tutti i client.
Evento 2:	La temperatura della CPU di "client1" ha superato la soglia critica.
Azione 2:	Avvisa gli amministratori via e-mail.
Evento 3:	Sono le ore 18:30.
Azione 3:	Comanda lo spegnimento di tutti i client di classe workstation.

Tabella 3.1: Alcuni esempi di eventi e relative azioni.

L'amministratore può implementare le azioni come ritiene opportuno per mezzo di plugins.

3.7 Interazione con un Batch System

L'obiettivo principale per il quale il prototipo è stato sviluppato è quello dell'interazione con un sistema di batch e scheduling per offrire in modo dinamico le piattaforme necessarie all'esecuzione dei job. Il prototipo è in grado di monitorare i job in coda presso il sistema e comandare l'avvio di determinai ambienti virtuali su un pool di Dom 0.

In figura 3.5 è mostrato uno schema esemplificativo dei meccanismi necessari allo svolgimento di tale compito da parte delle componenti del pro-

totipo. Essa mostra lo schema di un cluster ideale dove sui worker node è stato installato il sistema di virtualizzazione Xen.

Il manager riceve dai client le proprietà relative allo stato delle risorse dei worker node. Ciò comporta l'esecuzione dei client sui Dom 0 così da permettere il monitoraggio sia delle risorse hardware reali (poiché il Dom 0 gode di un accesso privilegiato e quindi trasparente al sistema di virtualizzazione) sia la presenza e lo stato delle eventuali macchine virtuali in esecuzione.

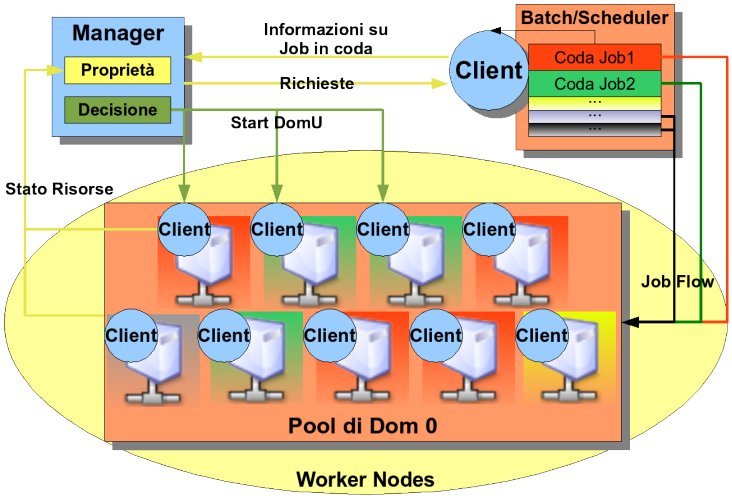


Figura 3.5: Interazione del prototipo con un batch system.

Un altro client in esecuzione sul batch system permette di esportare presso il Manager le informazioni relative alle code dello scheduler. Ciò consente di configurare il sistema di decisione del manager dimodoché prenda in considerazione le richieste d'esecuzione presso lo scheduler.

Nella figura sono evidenziate due code relative ad altrettante distinte tipologie di job. Il comportamento dello scheduler non è in alcun modo alterato, esso infatti continua a gestire il job flow, ovvero il commissionamento sui worker node delle richieste d'esecuzione, come in un normale cluster, ma in realtà i worker node sono dei Dom U.

Per mezzo della definizione di eventi il Manager è in grado di comandare attraverso i client l'esecuzione degli ambienti virtuali basandosi sull'entità del carico richiesto dai job in attesa sottomessi allo scheduler.

Il problema esposto in 3.2.1 può essere potenzialmente risolto in quanto l'esecuzione dinamica degli ambienti virtuali consente un partizionamento dinamico delle risorse, che permette una migliore gestione dei job, riducendo

i tempi d'attesa per essere eseguiti, e nel contempo uno sfruttamento migliore dei worker node, poiché ognuno di essi può essere utilizzato per l'esecuzione di una qualunque classe di job con l'unica limitazione della disponibilità delle risorse.

Inoltre gli ambienti virtuali vengono gestiti in modo del tutto automatico dal sistema, sgravando gli amministratori dalla configurazione manuale dei worker node. Infatti gli oneri di amministrazione si riducono al confezionamento degli ambienti virtuali per ogni classe di job tramite l'installazione delle immagini dei sistemi operativi e del software richiesto e della definizione dei parametri necessari agli ambienti virtuali (RAM, numero CPU, indirizzo IP, ecc.).

Non è escluso che si possano eseguire più ambienti virtuali sul medesimo worker node. Ciò è auspicabile ad esempio per l'esecuzione di determinati job seriali. Molti worker node infatti sono ricavati da macchine SMP, ovvero con più processori, che nell'esecuzione di job non parallelizzabili non verrebbero sfruttati appieno.

Xen consente di legare una macchina virtuale ad un processore specifico attraverso l'astrazione delle "virtual CPU". Eseguire più ambienti virtuali consente quindi di moltiplicare dinamicamente i nodi disponibili, senza aggiungere altro hardware.

Infine la definizione di specifiche classi di client consente di specificare quali ambienti virtuali un worker node può eseguire consentendo un assegnamento granulare delle risorse.

Capitolo 4

Prototipo: Realizzazione

4.1 Introduzione

Si vuole ora presentare la realizzazione del prototipo esposto nel capitolo precedente. Si mostrerà in principio l'ambiente di test sul quale il prototipo è stato realizzato. In seguito verranno illustrate le componenti del Manager, del client e della struttura di comunicazione, insieme a fornire l'implementazione dei meccanismi di funzionamento del prototipo.

4.2 Ambiente di Test

Per la realizzazione del prototipo e dei suoi successivi test si necessitava di un sistema che potesse riprodurre le caratteristiche di un ambiente distribuito. Di seguito sono fornite le specifiche per la realizzazione di tale ambiente all'interno del Laboratorio di Informatica del Dipartimento di Fisica dell'Università degli Studi di Perugia.

4.2.1 Il Laboratorio

Nel laboratorio sono presenti 35 postazioni adibite a workstation GNU/Linux.

Nel normale periodo di utilizzo didattico, ognuna di esse è impiegata come client disk-less, ovvero non ospita l'immagine del sistema operativo sul proprio hard-disk, bensì fa da terminale per un unico server X[1] in esecuzione sulla macchina con nome host *infolab* mostrata in figura 4.1, appositamente configurata per questo scopo.

[1] X Window System, noto anche con X11 o ancor più semplicemente X, è di fatto il gestore grafico standard per tutti i sistemi Unix.

Il sistema impiegato è LTSP (Linux Terminal Server Project) e consente il caricamento in RAM di un'immagine minimale del sistema operativo, sufficiente al funzionamento delle interfacce di rete e I/O necessarie agli utenti (Tastiera, Scheda Video, Mouse, ecc.). Agli utenti vengono fornite le comuni interfacce presenti su un desktop GNU/Linux (Nel caso specifico il Desktop Environment GNOME), ma in realtà l'esecuzione dei comandi avviene sul server, che gestisce anche l'autenticazione degli account.

Tutte le macchine sono connesse al medesimo segmento di rete e possono accedere alla rete dipartimentale e quindi ad internet attraverso il gateway *weblab*.

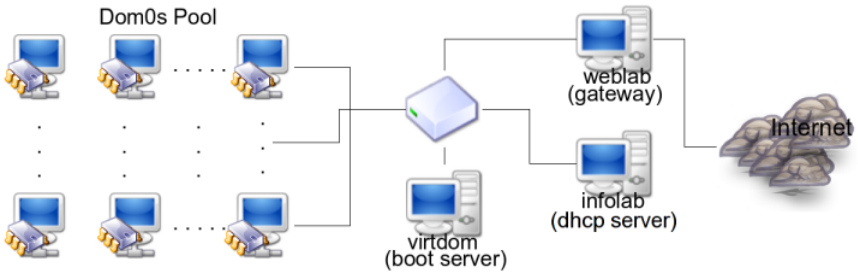


Figura 4.1: Schema del Laboratorio di Informatica presso il Dipartimento di Fisica dell'Università degli Studi di Perugia, impiegato come ambiente di test.

4.2.2 Pool di Dom0 su NFS

Per impiegare il laboratorio come ambiente di test è stato cruciale mantenere una separazione dell'utilizzo didattico da quello adatto ai nostri scopi, in modo che non interferissero uno con l'altro.

Per questo scopo e sfruttando la già presente infrastruttura che consente alle workstation il bootstrap via rete[2], è stato configurato un altro server, *virtdom* (su macchina virtuale), che fornisse le immagini dei sistemi operativi adatti a impiegare le workstation come Dom0.

Per diminuire il numero di installazioni da effettuare l'immagine dei Dom0 viene fornita attraverso export NFS. Ciò costituisce una novità nell'esecuzione dei domini privilegiati Xen. Ciò ha consentito di disporre di un pool di Dom0 che utilizza in numero variabile le workstation del laboratorio, sen-

[2]È il processo attraverso cui un computer esegue il proprio sistema operativo, caricandolo attraverso la rete piuttosto che da un supporto locale. Ciò può essere ottenuto attraverso la cooperazione di vari protocolli e componenti (DHCP, TFTP, PXE). Maggiori dettagli sono presenti in Appendice A

za dover interrompere la possibilità di fruizione dei normali servizi didattici offerti.

In un ottica più lungimirante, le macchine del laboratorio potranno a tutti gli effetti essere impiegate per l'esecuzione di ambienti virtuali afferenti al cluster stesso o a qualsivoglia altro compito per il quale si possano definire specifici ambienti virtuali.

Per maggiori dettagli si rimanda all'Appendice A.

4.3 Scelte di Implementazione

Prima di discutere nel dettaglio l'architettura del prototipo in questo paragrafo sono esposte alcune considerazioni sulle motivazioni riguardo le scelte implementative seguite.

4.3.1 Il Linguaggio di Programmazione

In primo luogo, il linguaggio scelto è stato da subito *Python*. Esso è un linguaggio di programmazione ad alto livello semi-interpretato, sviluppato a partire dal 1991 seguendo un approccio open source, che supporta diversi paradigmi di programmazione (Procedurale, Object Oriented, Funzionale). A motivare la scelta vi sono le seguenti caratteristiche:

- Codice facilmente leggibile che garantisce un'alta produttività.
- Efficenza maggiore rispetto ad altri linguaggi semi-interpretati.
- Ampio corredo di librerie e supporto all'integrazione di librerie di terze parti.
- È presente nel corredo software di tutte le distribuzioni GNU/Linux più utilizzate e di altri sistemi operativi.
- Programmazione Multi-threaded.

Python ha conosciuto una larghissima diffusione, e viene correntemente utilizzato sia come linguaggio di scripting che per l'implementazione di complesse applicazioni distribuite.

4.3.2 Virtualizzazione

Per il sistema di virtualizzazione è stato impiegato Xen. Insieme ai vantaggi già esposti, ve ne sono altri che l'hanno fatto preferire:

- Già ampiamente utilizzato presso il Dipartimento di Fisica.
- Degrado prestazionale trascurabile in molte applicazioni e minimo in altre.
- Il progetto viene continuamente aggiornato dalla comunità degli sviluppatori.

4.3.3 Comunicazione

Il sistema di comunicazione deve supportare la possibilità di inviare messaggi in multicast e punto-punto. Ciò implica che deve essere possibile la definizione di gruppi all'interno della rete. Inoltre per l'impiego in un ambiente distribuito è cruciale che le performance dello scambio dei messaggi siano elevate. Il *Toolkit Spread* fornisce tutte queste caratteristiche insieme a un'elevata tolleranza agli errori. Esso consiste in un pacchetto software distribuito da Spread Concepts e realizzato con la collaborazione di vari enti di ricerca e viene denominato un "sistema di comunicazione di gruppo".

4.4 Il Protocollo di Comunicazione

Spread può essere facilmente utilizzato attraverso una potente interfaccia di programmazione per la definizione di protocolli a livello applicazione.

Nel prototipo esso è impiegato per la realizzazione del protocollo di comunicazione che consente lo scambio di messaggi tra client e manager. In particolare, attraverso la rete devono passare: i comandi inviati dal manager verso i client, i messaggi relativi all'appartenenza ad un gruppo, le proprietà esportate e le risposte provenienti dai client.

Spread ha molte caratteristiche ma in questo paragrafo ci si limiterà a quelle rilevanti per la realizzazione del prototipo con riferimento al wrapper Python SpreadModule 1.5final e alla libreria Spread versione 3.17.4.

4.4.1 Panoramica di Spread Toolkit

Lo Spread Toolkit consta di un demone server che implementa le funzioni di gestione dei messaggi e di una libreria da utilizzarsi per la realizzazione delle applicazioni client. Il demone spread gestisce a basso livello la comunicazione tra le varie connessioni, implementando funzioni che consentono l'ordinamento del flusso dei messaggi, l'invio in multicast, la gestione dei gruppi e l'affidabilità nella consegna dei messaggi.

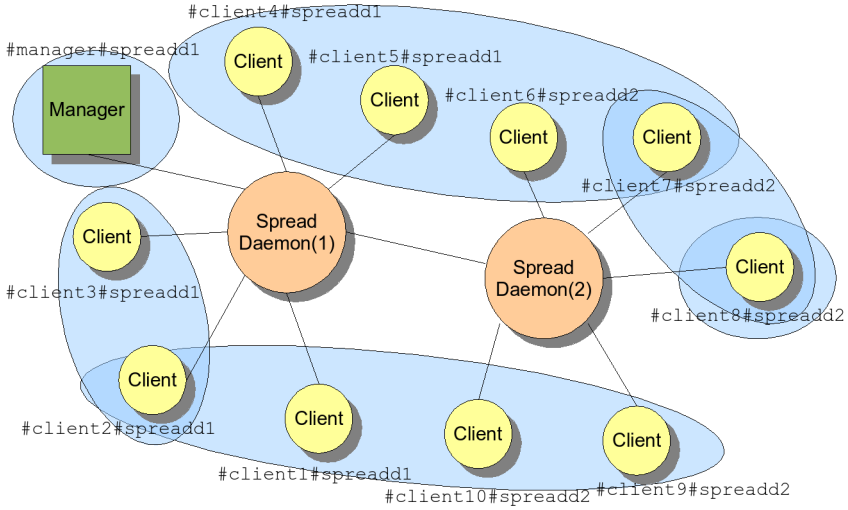


Figura 4.2: Rappresentazione di una rete spread con due demoni in relazione al prototipo.

L'API (Application Programming Interface) della libreria client fornisce un sapientemente ridotto set di funzioni attraverso cui è possibile costruire applicazioni con un alto grado di flessibilità. L'interfaccia, scritta in C, è supportata tramite wrapper da molti linguaggi tra i quali C++, Java, Ruby e ovviamente il wrapper Python al quale si fa riferimento.

In uno scenario di utilizzo tipico, come quello schematizzato in figura 4.2, i demoni in esecuzione possono essere anche più di uno e ognuno di essi è in grado di gestire un elevato numero di processi client in esecuzione sulla rete.

Le caratteristiche di ordinamento e affidabilità possono essere configurate per ogni messaggio, in modo da poter bilanciare performance e qualità del servizio secondo le proprie necessità.

Ogni client ha associato un nome spread, composto da una stringa alfanumerica non superiore a 32 caratteri. Ciascun client deve essere associato ad uno dei demoni spread in esecuzione sulla rete. Ci si può riferire univocamente ad un processo client attraverso una stringa del tipo:

```
#nome_spread_client#hostname_del_demone_spread
```

Tale nome può essere impiegato anche per l'invio di messaggi point to point specificandolo come destinatario. Saranno i demoni spread a gestire l'indirizzamento a più basso livello.

Sia il manager che i client gestiranno la propria componente di comunicazione attraverso una connessione spread. Ciò fa di loro dei client rispetto ai demoni spread.

In figura 4.2 è mostrato un esempio di rete spread che impiega due demoni per interconnettere tra loro un manager e più client. Le aree mostrate in azzurro rappresentano i gruppi definiti sulla rete. Vi sono gruppi che contengono un solo elemento come ad esempio quello che contiene il manager e uno di quelli che contengono il "client8". Per contro vi sono gruppi che contengono diversi client e alcuni client appartengono a più gruppi contemporaneamente.

4.4.2 Messaggi

Spread prevede due tipologie di messaggi: *Membership Messages* e *Regular Messages*. Entrambi sono caratterizzati da un diverso formato con differenti attributi.

Per ogni messaggio si può stabilire il grado di affidabilità e ordinamento con il quale si vuole che venga trattato dal sotto-sistema spread. Ciò consente di non doversi preoccupare delle perdite dei messaggi, sarà infatti il demone spread a farlo a seconda dell'affidabilità richiesta. Allo stesso modo l'invio dei messaggi gestiti dal demon spread può avvenire con ordinamento totale, il che, insieme alla garanzia di consegna, assicura che tutti i client ricevano i messaggi nel corretto ordine.

L'ulteriore strato d'astrazione introdotto dal wrapper Python semplifica l'interfaccia fornita dalla libreria spread fino a poter ignorare alcuni aspetti realizzativi di spread stesso. Di seguito non verranno descritti i messaggi spread, bensì la loro rappresentazione al livello d'astrazione del wrapper.

Messaggi Membership

I messaggi di Membership non vengono esplicitamente inviati dalle applicazioni, ma generati automaticamente quando un'applicazione client causa determinati eventi.

Nella tabella 4.1 sottostante è mostrata la rappresentazione di un messaggio membership secondo le specifiche del wrapper per Python e limitatamente ai campi rilevanti per il prototipo[3]

Il primo campo identifica il messaggio come di Membership ed è utilizzato dal wrapper per generare un oggetto corrispondente al messaggio. Ciò consente di avere un'astrazione del messaggio semplificando l'accesso ai campi

[3]Nell'implementazione originale di Spread Toolkit sono presenti altri campi che non vengono mostrati poiché non utilizzati o non rilevanti per l'applicazione.

campi	`MEMBERSHIP_MESSAGE`	`group`	`reason`	`extra`
tipo	*Intero*	*Stringa*	*Intero*	*Stringa*
significato	*Id Messaggio*	*Nome Gruppo*	*Id. Evento*	*Nome Client*

Tabella 4.1: Rappresentazione dei campi rilevanti per il prototipo di un messaggio Membership con riferimento all'astrazione fornita dal wrapper.

che diventano semplici attributi dell'oggetto stesso. L'estrazione delle informazioni rilevanti dai messaggi è gestita quindi a basso livello dal wrapper, sgravando il programmatore dalla necessità di gestire bit a bit il contenuto informativo del messaggio.

I campi successivi verrano quindi descritti assumendo questa caratteristica del wrapper, che consente di trattare le informazioni in essi contenute come tipi di dato del Python.

Il secondo campo, *group*, contiene la stringa identificativa del gruppo coinvolto nell'evento che ha causato l'invio del messaggio. Un gruppo viene infatti identificato da una stringa di massimo 32 caratteri, similmente ai nomi dei client.

I client possono associarsi e lasciare i gruppi per mezzo delle funzioni `join(nome_gruppo)` e `leave(nome_gruppo)`. Se il gruppo al quale un client richiede di associarsi non esiste, esso viene automaticamente creato. Tipicamente, al suo avvio il manager creerà tutti i gruppi necessari. Saranno poi gli algoritmi dei demoni spread a tenere traccia dell'appartenenza dei client ai gruppi.

Il campo *reason* specifica per mezzo di costanti numeriche il codice dell'evento che ha causato il messaggio. Gli eventi rappresentano i "join" e i "leave" da parte dei client. Ovvero il campo assume i valori `CAUSED_BY_JOIN` oppure `CAUSED_BY_LEAVE` a seconda se un client entra a far parte o lascia il gruppo specificato nel campo group. Accanto a questi due valori ve ne è un terzo importante per il prototipo: `CAUSED_BY_DISCONNECT`, che rappresenta la completa disconnessione della rete da parte di un client.

Infine il campo *extra* contiene la stringa che rappresenta il nome spread del client che ha causato l'evento di join, leave o disconnect. Il nome del client è specificato secondo il formato mostrato nel paragrafo 4.4.1.

I client spread possono essere configurati in modo tale da non essere inseriti tra i destinatari di messaggi di questo tipo e quindi da ignorarli, con una conseguente diminuzione del traffico sulla rete. I client del prototipo infatti non hanno necessità di ricevere messaggi di membership, sebbene è importante che essi li generino. Saranno quindi esclusi dalla ricezione. Per contro, attraverso questa tipologia di messaggi il nostro manager sarà in grado di conoscere lo stato di connessione dei client sulla rete e la loro appartenenza

ai gruppi definiti. Si vedranno ulteriori dettagli su questo aspetto quando si mostrerà la struttura interna del manager.

Messaggi Regular

Manager e client scambiano proprietà, comandi e altre informazioni attraverso Regular Messages. Per essi vale lo stesso discorso fatto nel precedente paragrafo riguardo ai messaggi di Membership, ovvero che il wrapper permette di astrarre rispetto al formato a basso livello dei messaggi e considerare direttamente i campi come attributi di un oggetto.

Essi sono mostrati nella tabella 4.2.

campi	`REGULAR_MESSAGE`	`groups`	`message`	`msg_type`	`sender`
tipo	*Intero*	*Lista*	*Stringa*	*Intero*	*Stringa*
significato	*Id Messaggio*	*Destinatari*	*"Payload"*	*Codice Mess.*	*Mittente*

Tabella 4.2: Rappresentazione dei campi rilevanti per il prototipo di un messaggio Regular con riferimento all'astrazione fornita dal wrapper.

Il primo campo è una costante numerica che identifica il tipo del messaggio e il suo impiego è del tutto analogo al caso dei messaggi di Membership.

Il campo *groups*, che occupa il secondo posto in figura, contiene la lista dei destinatari del messaggio. Esso può contenere sia nomi di gruppi che nomi di client o una combinazione dei due.

Il campo successivo, *message*, costituisce il cosiddetto "payload" del messaggio, ovvero il contenuto informativo vero e proprio. Esso è trasformato dal wrapper in una stringa e la sua lunghezza è limitata da spread a 100Kb.

msg_type contiene un intero a 16 bit (corrispondente al tipo `short` in C) che può essere specificato arbitrariamente dal mittente. Il manager e i client lo utilizzano per distinguere le varie categorie di messaggi che costituiscono il protocollo. A seconda del valore di questo campo, client e manager sono in grado di stabilire la semantica del messaggio ed estrarre dal contenuto informativo le informazioni.

Infine, il campo *sender* contiene il nome spread del mittente del messaggio.

4.4.3 Specifiche del Protocollo

Attraverso spread è stato definito un protocollo che consente il dialogo tra client e Manager. Sebbene il Manager rappresenti il nostro server nell'ambito del prototipo, nell'ottica di spread esso è semplicemente un client come gli altri. A tale proposito è importante introdurre il concetto di connessione spread. La creazione di una connessione spread avviene con la creazione di

un oggetto connessione attraverso la funzione `connect` fornita dal wrapper Python. Per ogni connessione va specificata la porta IP e il e il nome host del processo demone spread al quale ci si vuole connettere. Oltre a questi due parametri, ogni client può specificare il nome con il quale verrà identificato sulla rete e la volontà di ricevere messaggi di Membership. Come già accennato il manager sarà l'unico a ricevere questo tipo di messaggi.

L'oggetto restituito da connect viene detto *Mail Box* e permette l'invio e la ricezione di messaggi rispettivamente attraverso i metodi `multicast` e `receive`. In particolare multicast permette di specificare i campi dei messaggi Regular mostrati nel paragrafo precedente.

Il Manager possiede un set di messaggi utilizzato per l'interrogazione e il controllo dei client. Le categorie di messaggi sono le seguenti:

- Controllo della connessione dei client.
- Richiesta invio e aggiornamento proprietà.
- Esecuzione comandi.

Ognuno di essi è distinto da un particolare valore intero dell'attributo msg_type del messaggio regular, definite da costanti all'interno del programma. La semantica del contenuto informativo di ciascun messaggio è determinata dalla categoria a cui appartiene.

Tipicamente, ad ogni messaggio del manager, sia esso inviato in multicast oppure ad un destinatario specificato, corrisponde una risposta di *callback* dei client. Attraverso i callback i client restituiscono al manager le proprietà da esso richieste, oppure informano quest'ultimo della corretta esecuzione di un comando inviando il valore di ritorno. Le risposte dei client ricadono nelle categorie elencate:

- Messaggi informativi.
- Valori richiesti.
- Messaggi d'errore.

Messaggi del Manager

Di seguito vengono mostrati alcuni dei messaggi più importanti che costituiscono il dialogo tra client e manager. Verrà fatto riferimento ai campi dei messaggi Regular per mostrarne il significato. La descrizione assume che esista un client con nome "Client01" con il quale avviene il dialogo con il manager con nome "Manager".

campi	REGULAR_MESSAGE	groups	message	msg_type	sender
	Costante	*"Client01"*	*[proprietà]*	*GETPROP*	*"Manager"*
	Costante	*"Client01"*	*[proprietà]*	*FETCHPROP*	*"Manager"*

Tabella 4.3: Messaggi GETPROP e FETCHPROP.

GETPROP e FETCHPROP I messaggi *GETPROP* e *FETCHPROP* vengono impiegati dal manager per richiedere i valori e la lista delle proprietà esportate dai client. La tipologia dei messaggi è distinta da una costante numerica inserita dal mittente nel campo msg_type.

Una volta ricevuto il messaggio, e avendolo distinto in base alla costante numerica, il client reagisce diversamente a seconda del contenuto del campo message: se è esso è vuoto il messaggio viene interpretato come una richiesta di invio della lista delle proprietà. Se invece contiene una stringa essa viene interpretata come il nome della proprietà di cui il manager richiede il valore.

I messaggi *GETPROP* differiscono da quelli *FETCHPROP* poiché i primi richiedono il semplice invio delle proprietà mantenute dal client, mentre i secondi forzano l'aggiornamento delle proprietà prima dell'invio.

I client possono rispondere con i messaggi di callback PROPVAL, PROPLIST e PROPERR che verranno in seguito esposti.

campi	REGULAR_MESSAGE	groups	message	msg_type	sender
	Costante	*"Client01"*	*[comando [parametri]]*	*GETPROP*	*"Manager"*

Tabella 4.4: Messaggi EXECUTE.

EXECUTE Il manager richiede l'esecuzione di comandi presso i client per mezzo di messaggi con il campo msg_type impostato al valore della costante numerica EXECUTE.

Similmente al caso precedente, se non viene specificato nessun comando il client risponderà con la lista dei comandi disponibili. Altrimenti, se nel campo message è presente un nome di un comando conosciuto dal client e i suoi eventuali parametri, restituirà il valore di ritorno del comando.

I messaggi di callback che i client inviano in risposta possono essere EXECRET oppure EXECLIST.

campi	REGULAR_MESSAGE	groups	message	msg_type	sender
	Costante	*"Client01"*	*comando [parametri]*	*CONTROL*	*"Manager"*

Tabella 4.5: Messaggi CONTROL.

CONTROL Questa tipologia di messaggi viene utilizzata per gestire alcuni parametri di connessione client. Attraverso essi è possibile comandare la disconnessione di un client e la sua successiva immediata riconnessione. Il comando di controllo e i suoi eventuali parametri vengono specificati nel campo message. Alcuni di essi sono: `STOP` che causa la disconnessione del client, `RCON` per la reinizializzazione della connessione del client e `RJOIN <nome_gruppo>` che comanda ad un client di riassociarsi al gruppo specificato come parametro.

Callback dei client

In generale, ad ogni messaggio del manager ne corrisponde uno di callback da parte del client. In realtà quando non strettamente necessario alcuni messaggi di callback vengono evitati per diminuire il traffico sulla rete.

campi	REGULAR_MESSAGE	groups	message	msg_type	sender
	Costante	*"Manager"*	*"prop1":"type":"value"*	*PROPVAL*	*"Client01"*
	Costante	*"Manager"*	*"prop1","prop2",..*	*PROPLIST*	*"Client01"*
	Costante	*"Manager"*	*"propk"*	*PROPERR*	*"Client01"*

Tabella 4.6: Messaggi di callback relativi alle proprietà.

PROPVAL, PROPLIST e PROPERR I callback di risposta alle richieste delle proprietà da parte del manager sono i seguenti.

Il manager riceve la lista delle proprietà esportate da un client per mezzo del contenuto informativo dei messaggi di tipo `PROPLIST`, ciò consente al manager di conoscere quali proprietà un determinato client esporta e operare le proprie richieste di conseguenza. La lista è costituita dall'elenco delle etichette delle proprietà separate da un carattere separatore. La risposta a tali richieste è fornita dai client attraverso callback di tipo `PROPVAL`, nel campo message dei quali è contenuta l'etichetta, il corrispondente valore e il tipo (Intero, Stringa, ecc) della proprietà richiesta. Alla richiesta di una proprietà inesistente o non esportata da un client esso risponderà con un messaggio `PROPERR`, riportando l'etichetta della proprietà che ha generato l'errore.

campi	REGULAR_MESSAGE	groups	message	msg_type	sender
	Costante	*"Manager"*	*"cmd parametri":value*	*EXECRET*	*"Client01"*
	Costante	*"Manager"*	*"cmd1","cmd2",..*	*EXECLIST*	*"Client01"*

Tabella 4.7: Messaggi di callback relativi all'esecuzione di comandi.

EXECRET, EXECLIST Similmente alle risposte per le proprietà, i callback di tipo `EXECLIST` contengono nel campo message la lista dei comandi disponibili presso il client. I messaggi `EXECRET` restituiscono il risultato dell'esecuzione del comando richiesta dal manager oppure, in caso di errori il suo codice d'errore.

Callback Informative Affianco ai messaggi di callback mostrati ve ne sono altri nel set del client che costituiscono un insieme di risposte mirate a fornire al manager informazioni riguardo lo stato dei client. Tipicamente esse vengono fornite al manager in seguito alla ricezione di un messaggio `CONTROL` per semplificare la diagnostica di eventuali problemi.

4.4.4 Esempi di comunicazione

In questo paragrafo si vuole mostrare come avviene lo scambio di messaggi tra client e manager. Sebbene esso avvenga con la mediazione del demone spread, per semplicità la comunicazione viene mostrata in figura come se fosse diretta tra manager e client.

Il primo esempio mostra lo scambio di messaggi che avviene quando un client si affaccia sulla rete, effettua una connessione spread e entra a far parte di un determinato gruppo.

Nel secondo esempio è illustrato il caso in cui il manager sia costretto ad una riconnessione in seguito alla caduta della connessione.

Inizializzazione di un Client

Alla sua connessione il client entrerà a far parte di uno o più gruppi. Questo causerà la generazione di un messaggio di Membership che verrà ricevuto dal manager. A questo punto il manager si accorge della presenza di un nuovo client e può interrogarlo per conoscere le proprietà da esso esportate.

L'interrogazione avviene per mezzo di messaggi Regular `PROPVAL` richiedendo una per volta le proprietà nella lista inviata dal client in seguito ad una richiesta di tipo `PROPLIST`.

Caduta e Riconnessione del Manager

Nel caso in cui il manager dovesse venire a mancare, i client continuano indisturbati la propria esecuzione senza accorgersi dell'evento data la loro posizione subordinata rispetto al manager. Infatti essi inviano messaggi solo quando sollecitati a farlo dal manager stesso.

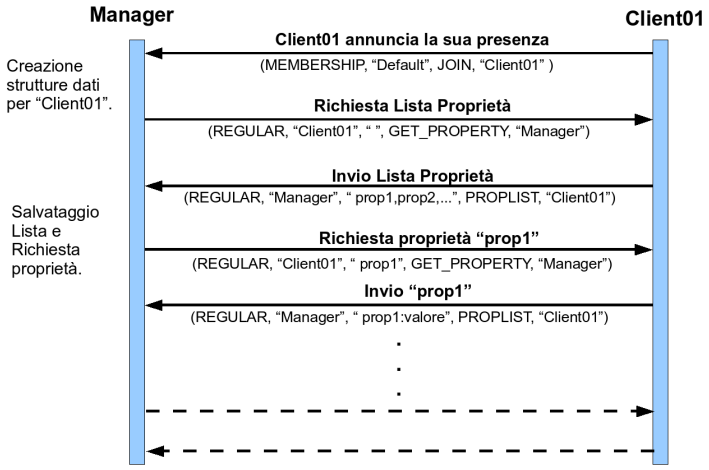


Figura 4.3: Inizializzazione di un client con nome “Client01” presso il Manager attraverso lo scambio di messaggi.

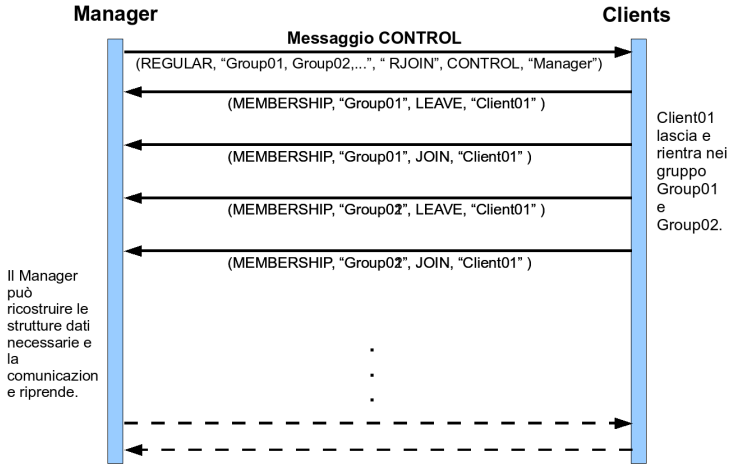


Figura 4.4: Scambio di messaggi successivo alla caduta e riconnessione del Manager.

Il manager, al momento della sua riconnessione invia in broadcast su tutti i gruppi di cui è a conoscenza un messaggio di tipo `CONTROL RJOIN`, dimodochè tutti i client rinnovino la propria associazione ai gruppi, causando un messaggio di membership. I client sono così di nuovo visibili dal manager

e l'esecuzione può riprendere senza problemi.

4.5 Client

La struttura della componente client è già stata esposta nel capitolo precedente. In questo paragrafo ne sono raccolti i dettagli implementativi.

4.5.1 Il Modulo runner

Come si è detto i client devono avere la possibilità di eseguire operazioni comandate dal manager. Tale possibilità è inoltre fondamentale per la raccolta delle informazioni che costituiranno i valori delle proprietà. È quindi stato necessario creare un modulo preposto allo svolgimento di questo compito. Tuttavia, poiché il client è in esecuzione come demone con pieni privilegi all'interno del sistema operativo, è importante limitare l'esecuzione di comandi potenzialmente dannosi per il sistema in cui si trova.

Il modulo *runner*, rappresentato nello schema di figura 4.5, è capace di indicizzare tutti i file eseguibili all'interno di una lista di directory di programmi accessibili dal client definita dall'amministratore, limitando ad essi possibilità d'esecuzione.

Il modulo sfrutta i costrutti offerti dal python per restituire sotto forma di stringa il risultato dell'esecuzione di un programma esterno all'interno del client. Pyhton infatti consente di invocare un interprete di comandi shell attraverso cui è possibile sottoporre i comandi. Al modulo runner è sufficiente il nome dell'eseguibile e i suoi eventuali parametri per autorizzarne l'esecuzione e passarlo all'interprete che ne restituirà il risultato.

Questo approccio garantisce un'alta flessibilità poiché i programmi eseguiti non hanno limitazioni sulla loro tipologia. Essi possono infatti essere script, programmi, o anche semplici link simbolici a programmi già presenti sul sistema. In questo modo, ad esempio, si può utilizzare direttamente il tool *xm* offerto da Xen per dialogare con *xend*, oppure il comando POSIX *ps* per avere la lista dei processi in esecuzione sulla macchina.

Affianco alla possibilità di impiegare programmi già esistenti, il vero vantaggio è quello di poter utilizzare programmi definiti dagli amministratori che tengano conto delle peculiarità di ciascuna macchina.

Il modulo runner rappresenta il mezzo attraverso cui il client si interfaccia con il sistema operativo che lo ospita, che gli consente l'estrazione di informazioni e lo svolgimento delle operazioni comandate dal manager.

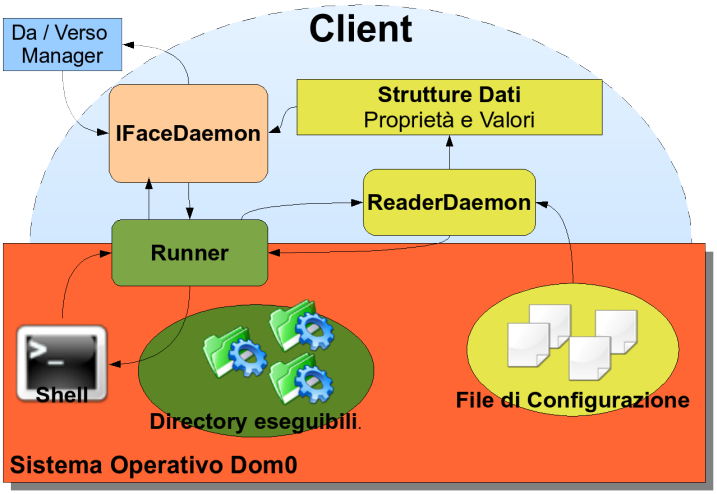


Figura 4.5: Schema dell'architettura del client con evidenziato il ruolo del modulo runner.

4.5.2 Gestione delle Proprietà

Nel paragrafo 3.4.3 si è visto che una proprietà è nient'altro che un etichetta associata ad un valore di interesse. Tali valori possono essere qualsiasi attributo del sistema sul quale il client è in esecuzione. Alcuni di questi attributi variano nel tempo, mentre altri rimangono costanti durante tutto il ciclo di vita del sistema. Ad esempio, la proprietà "memfree", già introdotta come esempio in 3.4.3, è una quantità molto variabile poiché dipende dalla quantità dei processi in esecuzione sul sistema. Al contrario, la proprietà "memtotal", che rappresenta la quantità di memoria RAM installata nel sistema, è ovviamente non soggetta a variazioni.

Le proprietà esposte come esempio sono corrispondenti a valori interi, ma non è tuttavia escluso che possano esistere valori di interesse di altro tipo. È il caso ad esempio della temperatura della CPU che è tipicamente un valore reale, oppure la stringa che identifica la partizione contenente il root filesystem, e molti altri ancora.

Alla luce di ciò, dal punto di vista del client, una proprietà è definibile attraverso i seguenti quattro attributi:

- Una stringa di etichetta, che consente il riferimento ad essa.
- Il comando con gli eventuali parametri da utilizzare per ottenere il valore della proprietà.

- Il tipo di dato che la proprietà rappresenta (Intero, Stringa, ecc.).
- Un intervallo di aggiornamento.

Il comando e i suoi parametri vengono passati al modulo runner per estrarre i valori corrispondenti e salvarle in apposite strutture dati. Tali strutture sono contenute nel client e sono gestiste come hash tables indicizzate sulle etichette delle proprietà. Ogni client avrà quindi una lista di definizioni di proprietà indicizzata sulle etichette che le identificano e una corrispondente lista dei valori di ognuno di esse.

Poiché i client non compiono operazioni di analisi e decisione basate sulle proprietà, essi si limitano alla semplice collezione dei valori. Sebbene sia specificato il tipo di ogni proprietà sarà il manager ad operare la distinzione, mentre nei client tutte le proprietà vengono trattate come stringhe.

Questa scelta facilita l'invio sulla rete delle proprietà in quanto il campo message dei messaggi regular deve essere specificato attraverso un parametro di tipo stringa dalla funzione multicast di spread.

La definizione delle proprietà avviene attraverso un formalismo introdotto in un file di configurazione che il client interpreta attraverso un parser all'inizio della sua esecuzione. Eccone un esempio:

```
[memfree]
command = ram
parameters = MemFree
return_type = int
ticks = 60
```

Tra parentesi quadre vi è l'etichetta della proprietà. Il comando `ram` è uno script che attraverso l'interrogazione dell'interfaccia offerta dai sistemi *unix-like* (*/proc/meminfo*) è in grado di estrarre informazioni sullo stato della memoria del computer. L'attributo `ticks` definisce l'intervallo di aggiornamento della proprietà.

Per l'implementazione dell'aggiornamento su base temporale delle proprietà, è necessario che venga definito un timer sul quale ci si possa basare nella definizione degli intervalli. Ciò è ottenuto attraverso la realizzazione di una classe detta *ThreadedDaemon* in grado di attivare, su un thread separato dal programma client principale, un ciclo generico d'esecuzione. L'attivazione di tale ciclo avviene in base a quanti di tempo definibili arbitrariamente.

La classe *ReaderDaemon* eredita da ThreadedDaemon questa caratteristica di esecuzione e basa l'intervallo d'aggiornamento delle proprietà su un quanto di tempo base denominato *poll_interval*. Tale quanto di tempo può essere definito dall'amministratore attraverso il file di configurazione del client.

Ogni proprietà, come mostrato nell'esempio, ha associato un attributo detto *tick*, che rappresenta un multiplo di poll_interval. Ciò consente di definire per ogni proprietà un intervallo di aggiornamento con la granularità desiderata.

ReaderDaemon ha il compito di esaminare le definizioni delle proprietà e invocare il loro aggiornamento attraverso il modulo runner, in accordo all'intervallo di aggiornamento base e quello definito per ogni singola proprietà.

4.5.3 Comunicazione con il Manager

Il client implementa il protocollo esposto nei precedenti paragrafi per mezzo di una classe di interfaccia denominata *IFaceDaemon*. Essa implementa un altro thread del tutto simile a ReaderDaemon in quanto anche essa eredita le sue caratteristiche da ThreadedDaemon.

La funzione principale è quella di restare in ascolto sulla rete spread e di gestire i messaggi provenienti dal manager in accordo con le specifiche definite dal protocollo.

In particolare, la ricezione di un messaggio viene implementata attraverso la funzione `receive()` offerta dal wrapper spread. Tale funzione è di solito bloccante, ovvero causa l'attesa indefinita del thread fino all'arrivo di un messaggio.

Affianco a questa funzione il wrapper mette a disposizione la funzione `poll()` che notifica la presenza di messaggi sull'oggetto MailBox generato dalla connessione spread. Con l'impiego di questa funzione IFaceDaemon può gestire la comunicazione controllando ad intervalli regolari la presenza di messaggi evitando il comportamento bloccante di `receive()`.

Il compito di IFaceDaemon non si esaurisce con la sola ricezione, bensì essa genera i messaggi di callback secondo le richieste inviate dal manager.

4.5.4 Gestione della Configurazione

La configurazione del client da parte dell'amministratore viene gestita attraverso più file di configurazione: uno principale e una lista di ulteriori file che contengono le definizioni delle proprietà.

La struttura del file principale segue lo standard de facto dei file di inizializzazione (tipicamente con estensione `.ini`, `.cfg` o `.config`) costituito da sezioni e parametri:

```
[Sezione1]
Parametro1 = valore
```

```
Parametro2 = valore
 ...
ParametroN = valore

[Sezione2]
Parametro1 = valore
Parametro2 = valore
 ...
ParametroN = valore

...

[SezioneN]
Parametro1 = valore
Parametro2 = valore
 ...
ParametroN = valore
```

Tra le sezioni quella denominata "*config*" contiene le impostazioni di inizializzazione del client. Il parser implementato nella classe *CConfigHandler* è in grado di distinguere più tipi di dato per ciascun valore e di tradurre direttamente in variabili ogni parametro. Esso inoltre interpreta tutte le altre sezioni come proprietà, secondo la definizione mostrata nel paragrafo precedente.

Di seguito è mostrato un esempio di configurazione:

```
[config]
;local config
exe_path = scripts/, progs/
fetch_interval = 1
user_properties = config_examples/client_user.cfg

;local spread config
spread_daemon = localhost
spread_port = 4803
local_name = client2
manager_name = manager
groups = default, testing

[proprietà1]
...
```

Il parametro `exe_path` definisce la lista di directory separate da virgola che contengono i file disponibili all'esecuzione da parte del modulo runner.

Il successivo `fetch_interval` permette di specificare in secondi l'intervallo base di aggiornamento delle proprietà.

`user_properties` contiene la lista di file di configurazione preposti alla definizione delle proprietà. Sebbene sia possibile definire proprietà all'interno del file di configurazione principale, la possibilità di aver più file permette di dividere in classi le configurazioni di proprietà secondo criteri che le accomuna logicamente.

I successivi parametri definiscono le impostazioni relative alla connessione spread. I parametri `spread_daemon spread_port` definiscono univocamente lo spread deamon al quale il client deve fare riferimento; inoltre, `local_name` contiene il nome spread con cui il client sarà conosciuto sulla rete. Queste informazioni diventeranno i parametri della funzione `connect()` impiegata dal wrapper per effettuare una connessione alla rete spread.

Il parametro `manager_name` definisce il nome spread del manager in esecuzione sulla rete al quale il client deve essere associato. Tale nome costituirà il contenuto del campo "*groups*", ovvero destinatario, di ogni messaggio in uscita dal client.

Infine, il parametro `groups` contiene la lista delle etichette di classe dei client. Ogni stringa contenuta in questo campo deve corrispondere ad un gruppo spread. Nella fase di inizializzazione e dopo aver creato la connessione spread, ogni client effettua il join in ciascun gruppo indicato dalle stringhe contenute in questo campo.

Il manager terrà traccia delle appartenenze dei client ai vari gruppi definiti per gestire la loro divisione in classi, e velocizzare la comunicazione.

4.6 Manager

Il manager è costruito seguendo un'architettura multithread per gestire i vari compiti ai quali è chiamato. In questo paragrafo si vedranno con maggior dettaglio le componenti del manager esposte nel capitolo precedente.

4.6.1 Strutture Dati

Il manager ha il compito di mantenere aggiornata una lista dei client connessi sulla rete e concentrare le informazioni da essi esportate come proprietà. Esso impiega diverse strutture dati per assolvere tali mansioni:

- Una classe detta *ClientDomain* le cui istanze rappresentano i client.

- La lista dei client connessi in rete.

- L'indice che raggruppa i client in classi corrispondenti ai gruppi spread.

Le liste delle proprietà, dei client, e dei gruppi sono implementate come tabelle hash.

L'implementazione di tali strutture dati è ottenuta impiegando il tipo di dato built-in "*dizionario*" offerto da python. I dizionari python sono considerabili come collezioni di dati eterogenei, indicizzati su chiavi anche esse eterogenee. Ciò è possibile poiché l'associazione tra chiavi e dati avviene attraverso una funzione hash calcolata in base all'*id* che l'interprete python stesso assegna ad ogni oggetto durante la sua inizializzazione. Poiché in python in generale ogni dato è un oggetto immutabile (stringhe, interi, ecc.), ognuno di essi ha un suo id e sarà quindi in base ad esso indicizzabile. In generale si può assumere che l'accesso puntuale ai dati avvenga con complessità O(1). Un'altra peculiarità dei dizionari python è la possibilità di iterare tra i suoi elementi come in una comune lista, consentendo quindi una scansione lineare completa degli oggetti che contiene.

La classe ClientDomain contiene dunque un dizionario che impiega le etichette delle proprietà come chiavi e associa ad ognuna di esse il corrispondente valore. Un'ulteriore struttura dati, una semplice lista concatenata, contiene l'elenco dei gruppi a cui è associato il client che l'istanza rappresenta. La classe offre poi i metodi d'interfaccia per implementare l'incapsulamento di tali strutture e gestire trasparentemente aggiornamenti e modifiche.

Nella classe *Manager*, che rappresenta la classe principale dell'applicazione, è contenuto il dizionario `clients` che fa corrispondere ai nomi spread dei client le astrazioni ottenute tramite le istanze di ClientDomain. Oltre a questo dizionario ve n'è un altro, identificato dalla variabile `groups`, indicizzato attraverso i nomi dei gruppi; gli elementi ad essi associati sono liste dei nomi dei client appartenenti ai gruppi stessi.

Il doppio indice così implementato consente di accedere efficientemente alle istanze ClientDomain sia in base al nome dei client sia attraverso la scansione lineare dei gruppi. Ciò consente di avere un'immediata corrispondenza tra le classi di client e i loro membri e, viceversa, tra i client e i gruppi spread a cui appartengono.

L'ovvio vantaggio, a fronte di una lieve ridondanza, è quello di poter accedere efficientemente a tutti i client di un gruppo, senza dover scandire linearmente il dizionario dei client e controllare la loro appartenenza. Considerando che molte delle decisioni prese dal manager sono basate sulla distinzione in classi dei client, questo vantaggio risulta essere molto conveniente.

Lo schema in figura 4.6 mostra lo schema delle strutture dati impiegate dal manager.

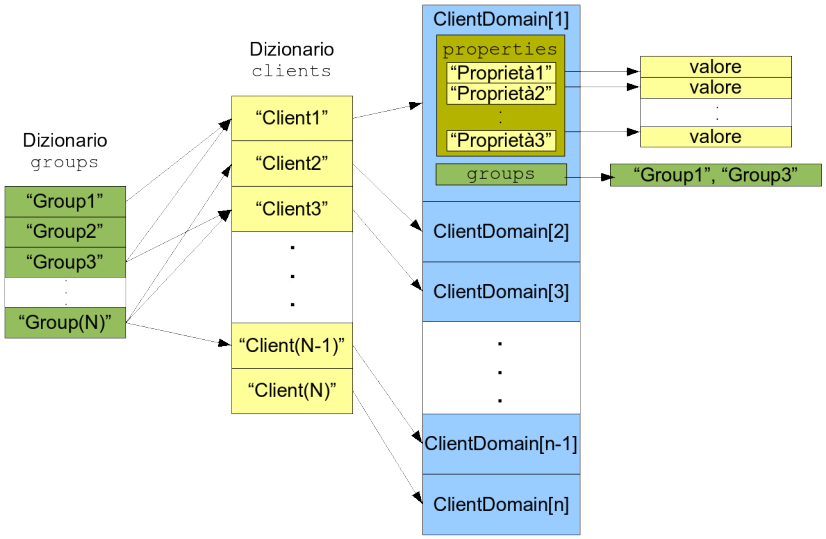


Figura 4.6: Rappresentazione delle strutture dati impiegate dal manager.

La gestione delle strutture esposte avviene in accordo agli eventi che si presentano sulla rete. Ad esempio il collegamento di un nuovo client e la sua successiva associazione ai gruppi, causa un messaggio di membership che farà inizializzare al manager un'istanza di ClientDomain, indicizzata secondo il nome corrispondente al nuovo client. In seguito il manager richiederà al client la lista delle proprietà che verrà salvata nell'apposito dizionario, in principio vuoto, all'interno dell'istanza del client. Poiché, come si è visto nei paragrafi precedenti, ogni associazione o abbandono di un gruppo da parte di un client causa la generazione di un corrispondente messaggio membership di join e leave, il manager sarà in grado di aggiornare contestualmente tutti gli indici basandosi sulla ricezione di tali messaggi.

Il comportamento seguito dal manager per la gestione delle strutture dati esposte è mostrato nel diagramma in figura 4.7.

Quelle esposte rappresentano le principali strutture dati implementate dal manager e gli consentono di avere una rappresentazione dello stato dei client presenti sulla rete. Tuttavia oltre ad esse, il manager ne implementa altre per la definizione e gestione del sistema di decisione e della con-

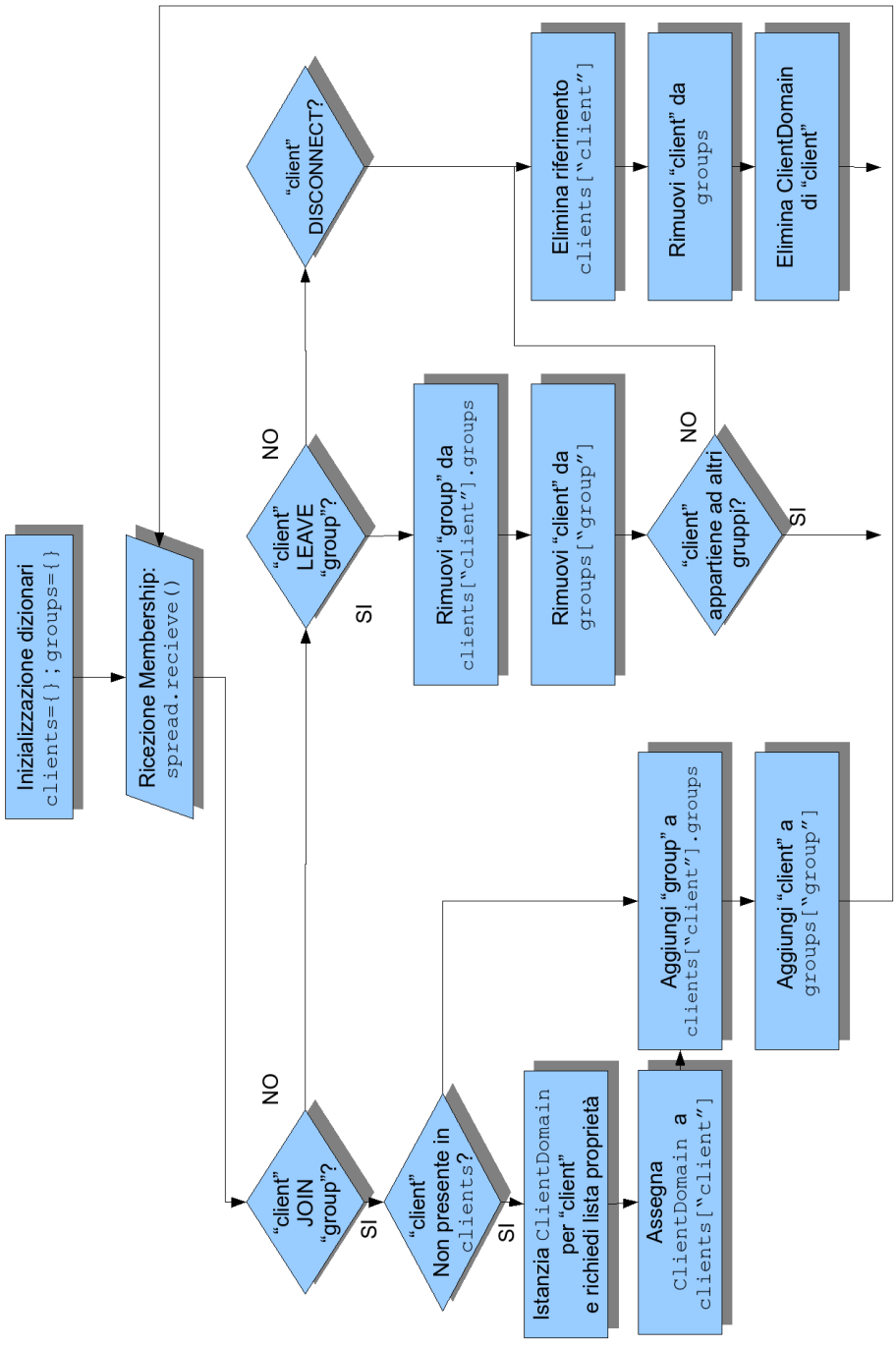


Figura 4.7: Diagramma di flusso del comportamento del manager per la gestione dei client.

figurazione. Queste strutture verranno introdotte e descritte nei successivi paragrafi contestualmente all'introduzione delle relative componenti.

4.6.2 Comunicazione con i Client

A differenza dei client il manager ha una parte attiva nella comunicazione, ovvero mentre i primi si limitano in modo subordinato a rispondere alle richieste, il manager genera messaggi di tipo regular per l'interrogazione e il controllo dei client.

L'inizializzazione della connessione spread avviene all'interno della classe manager che costituisce il thread d'esecuzione principale. Il riferimento al MailBox che essa genera e che consente l'invio e la ricezione dei messaggi, viene esportato a tutte le altre classi che necessitano della connessione.

Il protocollo di comunicazione esposto nei precedenti paragrafi è implementato all'interno di due classi separate. La prima di esse è la classe *Listener*, a cui è demandato il compito di gestire i messaggi regular e membership in ingresso. Analogamente a quanto visto per la classe IFaceDaemon dei client anche la classe Listener è una sotto-classe di ThreadedDaemon e dunque eredita la caratteristica di poter eseguire un ciclo d'esecuzione in un thread separato. In tale ciclo viene eseguita la funzione `receive()`, già vista in precedenza.

A differenza della connessione spread impiegata dai client, quella inizializzata dal manager gli permette di ricevere anche i messaggi di membership che, come è stato esposto, risultano di fondamentale importanza per la conoscenza dello stato della rete da parte del manager. La classe Listener ha il compito di richiamare i corretti metodi implementati dalla classe Manager per la gestione dei client in accordo con l'arrivo dei messaggi membership. Il suo contributo consente al manager di attivare il meccanismo di gestione delle strutture dati esposto in figura 4.7 e nel relativo paragrafo.

I messaggi regular in uscita possono essere inviati attraverso i metodi implementati dalla classe *Sender* che costituisce la seconda classe demandata alla gestione della comunicazione. Tali metodi rappresentano delle funzioni di comodo che gestiscono automaticamente alcuni dei parametri necessari all'invio dei messaggi definiti dal protocollo. Tuttavia, ognuno di essi si basa sul metodo generico `send()` di Sender, che fa a sua volta uso della funzione `multicast()` offerta dal wrapper.

Sebbene spread sia in grado di gestire automaticamente i messaggi, per avere un maggiore controllo sui messaggi in uscita è stata implementata la classe *QueuedSender* che eredita da Sender i metodi per l'invio dei messaggi e da ThreadedDaemon l'attivazione periodica.

A differenziare quest'altra classe da quella Sender vi è la diversa gestione dei messaggi. Essa è infatti implementata attraverso una coda FIFO controllata da un thread che si attiva ad intervalli configurabili. I messaggi generati non vengono immediatamente inviati bensì accodati e messi in attesa. È durante il ciclo d'esecuzione del thread che avviene l'effettiva chiamata al metodo `send()` che ne causa l'invio.

Questo accorgimento è utile a scongiurare il pericolo di un eccessivo e incondizionato traffico di messaggi generato dal manager: poiché nella classe Sender non vi è alcun controllo, i messaggi vengono immediatamente inviati e vi è quindi la possibilità che un eccessivo numero di messaggi incida fortemente sulle prestazioni di tutta la rete, saturandola; invece, attraverso l'attivazione periodica dell'invio dei messaggi, si garantisce un tempo minimo tra l'invio di un messaggio e l'altro.

4.6.3 Decisione: Trigger

Il manager invia ai client le sue richieste e i suoi comandi in base all'analisi delle informazioni che esso raccoglie riguardo lo stato della rete e le proprietà che i client esportano. In altre parole, il manager è in grado di decidere l'esecuzione di funzioni in base agli eventi che si presentano.

Un *evento* rappresenta quindi una contingenza nella quale una o più proprietà assumono un determinato valore che riveste un particolare significato ad un certo istante. Alcuni esempi di questo aspetto sono già stati mostrati in 3.6.

Un meccanismo che associ delle istruzioni agli eventi può essere dunque utilizzato per specificare i criteri di decisione che determinano il comportamento dell'intera applicazione.

Il meccanismo ad eventi del manager è costituito da tre elementi principali:

- Un insieme di direttive per la definizione degli eventi.
- Gli strumenti che permettono di definire le funzioni di gestione degli eventi, definiti *event-handler*.
- Un metodo principale, detto *event-dispatcher*, demandato al controllo del verificarsi degli eventi e attivi i corrispondenti event-handler.

Lo strumento che consente agli amministratori la definizione di eventi e degli event-handler è costituito dai *Trigger*. Simmetricamente, la definizione di un Trigger avviene specificando un evento e associando ad esso una funzione da eseguire quando l'evento si verifica.

Poiché si vuole che i Trigger siano definibili dagli amministratori attraverso la configurazione del manager, nasce l'esigenza di realizzare un linguaggio minimale, ma nel contempo abbastanza espressivo da poter offrire le direttive e gli strumenti per la definizione degli eventi e degli event-handler.

La realizzazione di un linguaggio porta tuttavia con se molte problematiche come ad esempio la definizione di una grammatica o la necessità di un parser.

Fortunatamente, python offre mezzi che consentono il parsing e l'interpretazione a run-time di porzioni di codice esterne al programma principale (a volte dette "plugins"). L'utilizzo di essi permette di sfruttare l'interprete python che è ovviamente già provvisto di un consistente sistema di parsing di una sintassi ben definita e potente.

I mezzi in questione sono costituiti dalle funzioni built-in `eval()` e `exec()`. La prima accetta come parametro un'espressione python e ne restituisce il valore. In modo analogo, la seconda esegue l'insieme di istruzioni passate per parametro sottoponendole all'interprete. A fianco a queste due, la funzione `compile()` restituisce l'oggetto che rappresenta il bytecode[4] ottenuto dalla compilazione di una stringa che rispetti la sintassi python passata per parametro.

Per costituire un Trigger sarà quindi sufficiente scrivere il codice python corrispondente alle espressioni che rappresentano gli eventi e le istruzioni che compongono le funzioni event-handler corrispondenti. Un positivo effetto collaterale di questo approccio è la possibilità di sfruttare trasparentemente la gestione degli errori e di controllo della sintassi implementate dall'interprete python. In seguito, l'utilizzo della funzione `compile()` e le caratteristiche del python, fanno sì che il codice scritto debba essere compilato una sola volta in bytecode, e poi poter essere eseguito attraverso `eval()` e `exec()`, saltando la fase di traduzione.

Il codice compilato viene conservato nelle strutture dati implementate dalla classe *Triggers*. Similmente alle proprietà dei client agli eventi viene associata un'etichetta attraverso la quale è possibile l'indicizzazione tramite un dizionario.

Il terzo elemento necessario alla gestione degli eventi, ovvero l'event-dispatcher, è implementato in due modi distinti. In primo luogo si può considerare l'arrivo dei messaggi dai client e il conseguente aggiornamento delle proprietà come una classe di eventi di livello superiore rispetto a quelli definiti dai Trigger. Tali aggiornamenti fanno scattare nel manager la valutazione dei

[4]Poiché python è un linguaggio semi-interpretato la fase di esecuzione deve essere preceduta da una di traduzione che generi il codice comprensibile all'interprete python, tale codice viene detto *bytecode*.

Trigger che sono stati configurati. In questo modo, l'esecuzione degli event-handler definiti per i trigger avviene immediatamente dopo l'aggiornamento, evitando la possibilità che essi basino le proprie valutazioni su proprietà non aggiornate e quindi non rispondenti alla reale situazione delle risorse.

L'altro event-dispatcher è costituito da un'altra classe ThreadedDaemon: la classe *TriggerWatch*. Essa legge le definizioni dei Trigger, valutando sequenzialmente ad intervalli regolari le espressioni che definiscono gli eventi e, per ogni espressione vera, esegue il corrispondente event-handler. Così come per le proprietà dei client, allo stesso modo per ogni operazione di "polling", ovvero l'operazione di valutazione del verificarsi degli eventi operata da un istanza di TriggerWatch, l'intervallo tra un esecuzione e la successiva può essere configurato in modo indipendente per ogni Trigger. Ulteriori dettagli sono contenuti nel paragrafo che segue.

Questa distinzione consente di effettuare le valutazioni dei Trigger che dipendono da valori locali al manager in modo indipendente rispetto a quelli che dipendono esclusivamente dai valori delle proprietà dei client.

4.6.4 Gestione della Configurazione

La configurazione del manager avviene per mezzo di file di configurazione in maniera del tutto analoga a quella del client. Anche in questo caso vi è la possibilità di definire più file di configurazione oltre a quello principale.

Il formato dei file è essenzialmente lo stesso di quello visto nel caso dei client, sebbene per la definizione dei Trigger si arricchisca di ulteriori costrutti.

Un esempio del file principale è il seguente:

```
[config]
spread_daemon = localhost
spread_port = 4803
local_name = manager
groups = default,another,some,iscsi
queued_sending = yes
queued_sending_interval = 0.1
triggers_eval_interval = 1
user_triggers = config_examples/manager_user.cfg

[client_properties_query]
expr = { len(manager.clients) != 0 }
cmd = { sender.send_get_property() }
on_update_check = no
```

```
ticks = 30

[client_dump]
expr = { True }
cmd = { for c in manager.clients.values():
      ----print c
    }
on_update_check = yes
ticks = -1


[test2]
expr = { len(manager.clients) == 0 }
cmd = ( file:../config_examples/trigger_test2 )
on_update_check = yes
ticks = -1
```

La sezione `config` contiene le impostazioni di inizializzazione del manager. Gli attributi `spread_daemon`, `spread_port` e `local_name` hanno lo stesso significato visto per i client: nome e porta del demone spread, il nome con il quale il manager sarà raggiungibile sulla rete spread. Il campo `group` contiene la lista di gruppi che il manager gestirà. Ciò consente di avere virtualmente più manager che gestiscono indipendentemente gruppi di client diversi e separati.

Gli attributi successivi, `queued_sending` e `queued_sending_interval` stabiliscono: il primo se si desidera utilizzare la coda FIFO per l'invio dei messaggi; il secondo l'intervallo di tempo espresso in secondi che deve trascorrere tra l'invio di un messaggio e un altro. Se il valore del primo è "no" il secondo viene ignorato.

Infine `trigger_eval_interval` rappresenta l'intervallo base di valutazione dei Triggers, e `user_trigger` definisce la lista dei file addizionali contenenti ulteriori definizioni di Trigger.

Ogni altra sezione viene interpretata come Trigger. Per ogni Trigger è necessario fornire nel campo `expr` l'espressione che definisce l'evento (o gli eventi) che esso rappresenta. Il campo `cmd` è utilizzato per la definizione dell'azione da intraprendere al verificarsi dell'evento corrispondente.

Le azioni possono essere definite direttamente scrivendo tra parentesi graffe il codice python che le implementa, oppure facendo precedere la stringa file *"file:"* al percorso di un modulo python contenuto in un file esterno.

Gli ultimi attributi da definire per la creazione di un Trigger riguardano la modalità di gestione, ovvero attraverso quale delle due tipologie di "event-

dispatching" implementate si vuole che vengano valutati gli eventi che esso definisce. Se il campo `ticks` contiene un valore positivo, esso viene interpretato come un multiplo dell'intervallo di valutazione base impiegato da TriggerWatch. Se invece contiene un numero negativo esso non verrà mai valutato attraverso TriggerWatch.

L'attributo `on_update_check` può essere impostato come vero o falso. Nel primo caso il Trigger verrà valutato ad ogni aggiornamento delle proprietà dei client; nel secondo si specifica la volontà di escludere il Trigger dalla valutazione in conseguenza agli aggiornamenti.

Capitolo 5

Conclusioni

La situazione attuale vede il prototipo in fase di attivo sviluppo. Sono stati già condotti diversi test per valutare la validità del protocollo di comunicazione e dell'interazione tra il manager e i client attraverso la definizione di proprietà e l'esecuzione di istruzioni comandate dal manager. L'ambiente di test presentato, caratterizzato da notevoli peculiarità nell'ambito delle realizzazione di sistemi che impiegano la para-virtualizzazione offerta da Xen, si è già rivelato di fondamentale importanza per l'implementazione di un testbed che consente di sperimentare le funzionalità del sistema.

Nel prossimo futuro si cercherà di sviluppare un set di script e programmi che consentiranno ai client di rispondere sia in modo generale alle più comuni necessità di implementazione di ambienti gestiti dinamicamente e sia relativamente all'interazione con un batch system. In particolare verranno creati strumenti in grado di gestire la sicronizzazione e la gestione delle immagini degli ambienti virtuali, fino al raggiungimento della definizione di metodologie standard per la replicazione degli ambienti in modo efficiente.

Xen permette inoltre di congelare lo stato delle macchine virtuali e la loro efficiente migrazione; si presentano quindi notevoli possibilità di espansione delle funzionalità del prototipo che si avvalgano di tali caratteristiche: meccanismi di checkpointing, gestione delle risorse con priorità e diritti di prelazione, e altro.

A questa fase seguirà la definizione di metriche per la valutazione dei benefici dovuti all'introduzione della gestione dinamica degli ambienti virtuali in sistemi batch, con l'obiettivo di poter testare il manager in un ambiente di produzione prima simulato e poi eventualmente reale (Il cluster INFN di Perugia).

L'interazione con un database è da prendere in considerazione poichè potrebbe semplificare notevolmente le strutture dati necessarie al manager e nel contempo fornire un accesso più efficiente a proprietà e stato dei client. In-

oltre la gestione di tipo transazionale garantisce vantaggi per quanto riguarda la consistenza dei dati e l'accesso concorrente ad essi.

Un'altro ambito di sviluppo è rappresentato dalla possibilità di integrare il prototipo con la gestione della rete del Dipartimento di Fisica che, attraverso l'impiego di tecnologie come le VLAN, i database LDAP, e le stesse metodologie di virtualizzazione Xen impiegate nel prototipo, consentirà una gestione del tutto dinamica dei servizi di rete, migrando su ambienti virtuali la maggior parte dei server e consentendo la definizione di meccanismi di alta affidabilità e disponibilità, sfruttando la già potente infrastruttura che ha già manifestato i suoi punti di forza.

Il lavoro svolto è stato estremamente stimolante e mi ha consentito di ampliare notevolmente le mie conoscenze nell'ambito dei sistemi distribuiti, della programmazione in rete e delle possibilità offerte dalle tecniche di virtualizzazione. Oltre a ciò mi ha dato la possibilità di dare uno sguardo abbastanza approfondito alle problematiche delle realtà che rappresentano le attuali frontiere dei sistemi di calcolo come i cluster di computer e le griglie computazionali. Inoltre ho potuto apprezzare le dinamiche del lavoro di gruppo, potendomi avvalere delle conoscenze di persone che hanno contribuito alla mia crescita sul piano delle competenze e ancor più su quello personale.

Ringraziamenti

Desidero ringraziare il Prof. Leonello Servoli per la sua enorme disponibilità, per l'opportunità offertami e per la sua guida durante tutto il percorso di tesi.

Un ringraziamento di cuore al grande Mirko Mariotti, per il suo supporto, amicizia e per le innumerevoli cose che mi ha insegnato.

Grazie a Claudio, Flavio, Francesco, Hassen, Igor e Simone con cui ho trascorso tutto il periodo di stage e a cui devo molto.

Grazie a mio fratello Marco, a Nello, Giuseppe e Luisa e a tutti i miei amici per il loro affetto, aiuto e supporto.

Un grazie, mai sufficiente ma che vale una vita, ai miei genitori, vero modello e sostegno, esteso a tutta la mia famiglia, in particolare Zio Antonello, Zia Silvana e le Nonne.

Grazie a tutti i giganti dell'informatica e delle scienze sulle cui spalle sono potuto salire, a volte immodestamente.

Grazie al tempo che scorre, ai tardi pomeriggi d'estate ed al rock'n'roll.

RMC

Appendice A

Realizzazione dell'Ambiente di Test

A.1 Introduzione

Di seguito sono esposte le procedure e configurazioni seguite per ottenere l'ambiente di test di cui si è parlato in 4.2 e che è stato impiegato per la realizzazione del proptotipo. Per la descrizione del laboratorio si rimanda a 4.2.1.

Per avere un'ambiente funzionante sono necessari vari servizi:

- Le postazioni devono essere equipaggiate con interfacce di rete che supportano PXE[1]
- Un server DHCP.
- Un server TFTP (Trivial File Transfer Protocol).
- Il bootloader GRUB (nella versione per PXE, ovvero pxeGRUB).
- Un server NFS.

I server DHCP e TFTP verranno ospitati da *infolab*. Poichè tutte le workstation si trovano sul medesimo segmento di rete, sarà sufficiente l'impiego di un singolo server DHCP. Infolab conterrà inoltre l'immagine del root filesystem da esportare via NFS per i Dom0.

[1]Preboot Execution Enviroment, viene impiegato per fornire un'ambiente d'esecuzione iniziale necessario al caricamento via rete del sistema operativo.

A.2 Interfacce di rete con PXE

Una postazione diskless necessita in primo luogo di un metodo per ottenere attraverso la rete il kernel da eseguire. Le interfacce di rete delle workstation del laboratorio sono state scelte in modo che potessero impiegare PXE. Ognuna di esse ha una memoria ROM che consente il caricamento dell'ambiente iniziale. Il boot via rete deve essere abilitato dal BIOS (abilitando caratteristiche del tipo: Booting Other Devices, LAN Boot Rom, ecc).

A.3 Configurazione DHCP e pxeGRUB

La prima interazione delle interfacce di rete delle workstation avviene per mezzo di una richiesta DHCP che viene raccolta dal server su infolab.

Le informazioni necessarie per il boot via rete sono specificate all'interno del file di configurazione del server DHCP. Oltre alle informazioni relative alla configurazione della rete, esso offre alle workstation l'indirizzo di un "boot server", il nome del bootloader e i file di configurazione ad esso necessari.

Nella tabella A.1 è riportato un esempio del file di configurazione impiegato.

Come accennato, attraverso il server DHCP è possibile fornire ad ogni macchina il file di configurazione del bootloader pxeGRUB. In questo modo si può stabilire il comportamento di ogni macchina semplicemente editando la linea "default" nel file di configurazione di pxeGRUB: Le postazioni possono procedere con il processo di boot per l'esecuzione del normale ambiente didattico LTSP attraverso *infolab*, oppure richiedere istruzioni ulteriori al server *virtdom*, che offre un altro file di configurazione per il bootloader fornendo le informazioni necessarie al boot come Dom0.

Questo permette di stabilire quali macchine debbano essere impiegate come Dom0 e quali altre devono essere impiegate come workstation per uso didattico. Per ogni macchina è quindi presente un distinto file di configurazione, come mostrato nella tabella A.2

A.4 Configurazione Boot Server (virtdom)

La macchina *virtdom* verrà impiegata per ospitare le risorse necessarie alle workstation che eseguono il boot dei Dom0. Queste risorse sono rappresentate da:

- Le immagini dell'Hypervisor Xen.
- Le immagini dei kernel modificati per essere impiegate come Dom0.

```
 [...]

option space PXE;
option PXE.mtftp-ip               code 1 = ip-address;
option PXE.mtftp-cport            code 2 = unsigned integer 16;
option PXE.mtftp-sport            code 3 = unsigned integer 16;
option PXE.mtftp-tmout            code 4 = unsigned integer 8;
option PXE.mtftp-delay            code 5 = unsigned integer 8;
option PXE.discovery-control      code 6 = unsigned integer 8;
option PXE.discovery-mcast-addr   code 7 = ip-address;

class "pxeclients"
{
        match if substring (option vendor-class-identifier, 0, 9) =
"PXEClient";
        option vendor-class-identifier "PXEClient";
        vendor-option-space PXE;

        option PXE.mtftp-ip 0.0.0.0;
}

[...]

option option-150 code 150 = text;
option option-128 code 128 = string;
option option-129 code 129 = text;

[...]

## Terminals

host post01 {
    next-server          192.168.0.100;
    hardware ethernet    00:11:2f:a8:b4:d4;
    fixed-address        192.168.0.101;
    filename             "pxegrub";
    option option-150    "/grub/post01.conf";
}

[...]
```

Tabella A.1: Esempio di file di configurazione del server DHCP.

```
default 0
timeout 2
password xxxxx

title   Infolab terminal
        root (nd)
        kernel /kernel/ltsp-2.6.17.8 rw root=/dev/ram0
        initrd /initrd/ltsp-2.6.17.8.gz
title   On-demand virtual machines
        password xxxxx
        dhcp
        tftpserver 192.168.0.5
        root (nd)
        configfile /grub/post01.conf
```

Tabella A.2: Esempio di file di configurazione di pxeGRUB.

- I file di configurazione dei bootloader di ogni singola macchina.
- Il filesystem di root.

Per offrire queste risorse *virtdom* impiega le seguenti tecniche:

- **TFTP** per le immagini dell'Hypervisor e del Kernel e i file di configurazione di GRUB.
- **NFS** per il filesystem di root.

A.4.1 TFTP

TFTP è un semplice protocollo per il trasferimento dei file che offre le funzionalità basilari di FTP. Data la sua caratteristica di essere molto semplice e quindi occupare una quantità di memoria estremamente limitata, si presta molto bene ad essere mantenuto nello stack PXE delle schede di rete (tipicamente contenuto sulle ROM) che dispongono di pochi kilobyte di memoria. È quindi l'ideale per l'esecuzione iniziale delle macchine diskless.

Di seguito è mostrato il file di configurazione del server TFTP.

```
#Defaults for tftpd-hpa
RUN_DAEMON="yes"
OPTIONS="-l -s /tftpboot"
```

Il file di configurazione definisce `/tftpboot` come "*base directory*" dalla quale verranno rese disponibili le risorse presso le workstation. Più nel dettaglio, la base directory conterrà le seguenti sotto-directory:

`/tftpboot/grub` che contiene i file di configurazione di GRUB necessari alle workstation.

`/tftpboot/hypervisors` dove sono conservate le immagini dello Xen Hypervisor.

`/tftpboot/kernels` conenente le immagini dei Kernel Dom0.

A.4.2 GRUB

Un tipico file di configurazione di GRUB ospitato su virtdom è esposto di seguito:

```
 #post-pIV-realtek - NFS root

default 0
timeout 2

title   Dom0 - NFS
        root (nd)
        kernel /hypervisors/xen-3.1.gz dom0_mem=32M
        module /kernels/kernel-2.6.20-xen-r6-pIV-rltk \
ip=dhcp root=/dev/nfs \
nfsroot=192.168.0.5:/opt/ondemand/root
```

Un file di configurazione simile è messo a disposizione per ogni workstation via TFTP attraverso la directory preposta a ciò. Tali file possono essere recuperati dalle workstation alla posizione `/tftpboot/grub/<hostname>.conf`. Tipicamente essi sono dei collegamenti simbolici ai file reali, nei quali si può specificare per ogni macchina Hypervisor, Kernel e root filesystem. Il corretto file di configurazione viene scelto attraverso il primo menu offerto da pxeGRUB.

A.4.3 NFS

La configurazione del server NFS ospitato da *virtdo* è relativamente semplice. Eccone il file `/etc/exports` che contiene la definizione delle directory esportate sulla rete:

```
 #virtdom exports
/opt/ondemand/root              192.168.0.0/255.255.255.0(ro,no_root_squash,async)
/opt/ondemand/imgs              192.168.0.0/255.255.255.0(rw,no_root_squash,async)
```

In `/opt/ondemand/root` vi è l'albero del root filesystem ottenuto dall'installazione di una distribuzione minimale *GNU/Linux Debian etch*. Il filesystem è propriamente configurato in modo da supportare l'esecuzione di un ambiente Xen 3.1.

Come si può notare il filesystem di root è esportato in modalità read only, in modo da poter essere condiviso tra le workstation senza che venga da una di esse accidentalmente modificato. Gli aggiornamenti, installazioni e modifiche ai file di configurazione del root filesystem possono essere fatte una volta per tutti i client attraverso un ambiente in *chroot* accessibile da *virtdom*. Verranno presentati ulteriori dettagli in seguito.

A.5 Configurazione delle Workstation

A.5.1 Configurazoine del Kernel

Il kernel impiegato è ovviamente Linux e oltre a necessitare dell'applicazione delle modifiche per essere eseguito come un dominio privilegiato xen, esso deve essere configurato in modo da permettere di utilizzare il filesystem attraverso NFS.

Seguono alcune delle configurazioni necessarie contenute nel file `.config`, utilizzato nel processo di compilazione del kernel.

- I driver delle schede di rete devono essere compilate nell'immagine kernel, a seconda della tipologia di scheda impiegata (in questo caso 3com):

  ```
  [...]

  CONFIG_NET_VENDOR_3COM=y
  CONFIG_VORTEX=y

  [...]
  ```

- L'opzione "*IP Kernel Level Autoconfiguration*" deve essere abilitata:

  ```
  [...]

  CONFIG_IP_PNP=y
  CONFIG_IP_PNP_DHCP=y

  [...]
  ```

- Supporto per "*Root over NFS*" ovvero per la possibilità di impiegare un filesystem ospitato su un export NFS:

```
 [...]

CONFIG_NFS_FS=y
CONFIG_NFS_V3=y
CONFIG_NFS_V3_ACL=y
# CONFIG_NFS_V4 is not set
CONFIG_NFS_DIRECTIO=y
CONFIG_NFSD=y
CONFIG_NFSD_V2_ACL=y
CONFIG_NFSD_V3=y
CONFIG_NFSD_V3_ACL=y
# CONFIG_NFSD_V4 is not set
CONFIG_NFSD_TCP=y
CONFIG_ROOT_NFS=y
CONFIG_NFS_ACL_SUPPORT=y
CONFIG_NFS_COMMON=y

[...]
```

- Configurazioni relative all'utlizzo del kernel come Dom0 Xen:

```
 [...]

CONFIG_X86_XEN=y
# CONFIG_PCI_GOXEN_FE is not set
CONFIG_XEN_PCIDEV_FRONTEND=y
# CONFIG_XEN_PCIDEV_FE_DEBUG is not set
# CONFIG_NETXEN_NIC is not set
CONFIG_XEN=y
CONFIG_XEN_INTERFACE_VERSION=0x00030205
# XEN
CONFIG_XEN_PRIVILEGED_GUEST=y
# CONFIG_XEN_UNPRIVILEGED_GUEST is not set
CONFIG_XEN_PRIVCMD=y
CONFIG_XEN_XENBUS_DEV=y
CONFIG_XEN_BACKEND=y
CONFIG_XEN_BLKDEV_BACKEND=y
CONFIG_XEN_BLKDEV_TAP=y
```

```
CONFIG_XEN_NETDEV_BACKEND=y
# CONFIG_XEN_NETDEV_PIPELINED_TRANSMITTER is not set
CONFIG_XEN_NETDEV_LOOPBACK=m
CONFIG_XEN_PCIDEV_BACKEND=y
CONFIG_XEN_PCIDEV_BACKEND_VPCI=y
# CONFIG_XEN_PCIDEV_BACKEND_PASS is not set
# CONFIG_XEN_PCIDEV_BACKEND_SLOT is not set
# CONFIG_XEN_PCIDEV_BE_DEBUG is not set
CONFIG_XEN_TPMDEV_BACKEND=m
CONFIG_XEN_BLKDEV_FRONTEND=m
CONFIG_XEN_NETDEV_FRONTEND=m
CONFIG_XEN_SCRUB_PAGES=y
CONFIG_XEN_DISABLE_SERIAL=y
CONFIG_XEN_SYSFS=m
CONFIG_XEN_COMPAT_030002_AND_LATER=y
# CONFIG_XEN_COMPAT_030004_AND_LATER is not set
# CONFIG_XEN_COMPAT_LATEST_ONLY is not set
CONFIG_XEN_COMPAT=0x030002

[...]
```

A.6 Root filesystem via NFS

Il filesystem impiegato per le workstation è ottenuto a partire da una installazione minimale della distribuzione GNU/Linux Debian etch con aggiunti gli strumenti software *udev*, *python*, *bridge tools* e *xen tools* installati dai sorgenti.

Come si è detto il filesystem viene esportato via NFS da virtdom in modalità *read-only*. Tuttavia per una corretta esecuzione alcune porzioni dei esso devono essere scrivibili dalle workstations (ad esempio `/var/run`, `/var/lock`, ecc.). Per questo motivo, attraverso uno script eseguito in fase di boot, il sistema operativo creerà un albero di directory che contiene le locazioni che devono essere accessibili in scrittura. In seguito questo albero di directory verrà montato in una directory detta `/flash`. Nel filesystem di root comune tutti i file e le directory che devono essere scrivibili devono essere spostati in `/flash` e rimpiazzati con link simbolici nelle loro posizioni originali. Ciò permette di mantenere la consistenza dei link simbolici sia nell'ambiente chroot accessibile da virtdom, sia nel filesystme creato in fase di boot dalle workstation.

Infatti, nell'ambiente chroot, i link simbolici punteranno alle corrette posizioni spostate in `/flash`. In modo analogo, le workstation monteranno la porzione scrivibile del filesystem sulla directory `/flash` e l'albero delle directory comune contenuto in `/flash` verrà rimpiazzato verrà rimpiazzato in fase di boot con quello costruito in RAM da ogni workstation.

Il risultato delle modifiche apportate al filesystem è riportato nei seguenti listati:

L'export `/opt/ondemand/root` costituisce il root filesystem esportato presso le workstation.

```
 virtdom:~# ls /opt/ondemand/root/
bin  boot  dev  etc  flash  home initrd  lib
media mnt opt  proc root  sbin  srv  sys  tmp
usr  var
```

I file reali vengono spostati in `/flash` e sostituendoli nelle loro posizioni originali da collegamenti simbolici che puntano ad essi in `/flash`:

```
 virtdom:~# ls -l /opt/ondemand/root/flash/*
/opt/ondemand/root/flash/etc:
total 4
drwxr-xr-x 4 root root 4096 Nov 29 12:18 udev

/opt/ondemand/root/flash/var:
total 20
drwxr-xr-x 5 root root 4096 Nov 26 10:59 lib
drwxrwxrwt 2 root root 4096 Nov 16 13:07 lock
drwxr-xr-x 5 root root 4096 Nov 16 12:56 log
drwxr-xr-x 2 root root 4096 Nov 21 17:27 run
drwxrwxrwt 2 root root 4096 Nov 23 16:37 tmp

virtdom:~# ls -l /opt/ondemand/root/var/
[...]
drwxrwsr-x  2 root staff 4096 Oct 28  2006 local
lrwxrwxrwx  1 root root    15 Nov 16 18:37 lock -> /flash/var/lock
lrwxrwxrwx  1 root root    14 Nov 16 18:37 log -> /flash/var/log
drwxrwsr-x  2 root mail  4096 Oct 29 15:59 mail
drwxr-xr-x  2 root root  4096 Oct 29 15:59 opt
lrwxrwxrwx  1 root root    14 Nov 16 13:06 run -> /flash/var/run
drwxr-xr-x  3 root root  4096 Oct 29 16:01 spool
drwxr-xr-x  3 root root  4096 Nov 16 09:42 xen
```

Su una generica workstation, il contenuto di `/var` sarà identico a quello contenuto in `/opt/ondemand/root/var` contenuto nella directory di export NFS sul server virtdom:

```
 post23:~# ls -l /var/
```

```
[...]
drwxrwsr-x  2 root staff 4096 Oct 28  2006 local
lrwxrwxrwx  1 root root    15 Nov 16 18:37 lock -> /flash/var/lock
lrwxrwxrwx  1 root root    14 Nov 16 18:37 log -> /flash/var/log
drwxrwsr-x  2 root mail  4096 Oct 29 15:59 mail
drwxr-xr-x  2 root root  4096 Oct 29 15:59 opt
lrwxrwxrwx  1 root root    14 Nov 16 13:06 run -> /flash/var/run
drwxr-xr-x  3 root root  4096 Oct 29 16:01 spool
drwxr-xr-x  3 root root  4096 Nov 16 09:42 xen
```

Differentemente, la directory */flash* conterrà l'albero scrivibile generato separatamente in RAM da ogni client.

```
 post23:~# ls -al /flash/
total 4
drwxrwxrwt  4 root root   80 Jan 18 11:34 .
drwxr-xr-x 20 root root 4096 Nov 26 09:46 ..
drwxr-xr-x  4 root root   80 Jan 18 11:34 etc
drwxr-xr-x  7 root root  140 Jan 18 11:34 var
```

Il risultato d'insieme è quello voluto, ovvero effettuando un chroot da virtdom sul filesystem esportato, esso sembrerà identico a quello mostrato sulle workstation, ma sarà completamente scrivibile. Invece, dalle workstation l'unica posizione scrivibile sarà contenuta in `/flash`, e tutto il resto del filesystem resterà in modalità read-only.

A.6.1 Init script

La creazione della porzione scrivibile di filesystem in RAM viene effettuata da ogni workstation tramite un init script appostitamente realizzato per questo scopo. Lo script viene richiamato dall'init system di Debian e quindi eseguito in fase di boot.

```
  #!/bin/sh
#
# rw filesystem initialization
#
#   Thu Nov 22 16:24:17 UTC 2007
#

#Temporary folder where the writable elements are going to be created
TMPBASE="/mnt/tmp"
#The location where the writable filesystem will be mounted
BASE="/flash"

DIRS="var \
    var/run \
```

```
    var/run/screen \
    var/log \
    var/lock \
    var/lib \
    var/lib/xend \
    var/lib/xenstored \
    var/lib/urandom \
    var/lib/dhcp3 \
    var/tmp\
    etc \
    etc/network \
    etc/network/run \
    etc/udev"

DEVBASE="/dev"

DEVDIRS="pts \
    shm"

WAIT=0.1


#generating devices files
mkdev() {
    mount -n -t tmpfs tmpfs ${DEVBASE}

    for i in $DEVDIRS ; do
        mkdir ${DEVBASE}/${i}
    done

    for i in `seq 1 6`; do
        mknod /dev/tty$i c 4 $i
    done

    mknod /dev/tty c 5 0
    mknod /dev/null c 1 3
    mknod /dev/console c 5 1
    mknod /dev/random c 1 8
    mknod /dev/urandom c 1 9
}

#creates the filesystem in the temporary location
mktmp() {
    #the filesystem is created in ram
    mount -n -t tmpfs tmpfs ${TMPBASE}

    for i in $DIRS ; do
        mkdir ${TMPBASE}/${i}
        chmod +w ${TMPBASE}/${i}
```

```
    done
    rsync -a /flash/etc/udev ${TMPBASE}/etc
}

#at the end we move the newly created stuff in the proper location
domove() {
    mount --move ${TMPBASE} ${BASE}
}

#"local" commands
dolocal() {
    touch ${BASE}/var/log/dmesg
    echo "lo=lo" > ${BASE}/etc/network/run/ifstate
}

#what to do
DOLIST="mkdev \
    mktmp \
    domove \
    dolocal"

doit() {
    SN=‘echo -n $0 | sed 's:^/.*/::’‘

    for i in $DOLIST ; do
        echo -n "${SN}: "
        echo $i
        $i
    done;

    echo -n "${SN}: "
    echo "$WAIT Seconds..."
    sleep $WAIT
}

#do it!
doit
```

A.7 Modifica degli script di Xen

Nell'operazione di virtualizzazione delle periferiche di rete, Xen modifica il nome dell'interfaccia di rete ethernet fisica da "`eth0`" in "`peth0`", crea una nuova interfaccia virtuale chiamata "`eth0`" e trasferisce le informazioni di connessione (Indirizzo IP, Subnet Mask, ecc) ad essa, ovvero all'interfaccia che viene vista dal Dom0.

Questa operazione non crea generalmente grossi problemi, tuttavia, nel nostro caso, rende le workstation inutilizzabili poichè la condivisione NFS viene a mancare inaspettatamente essendo essa strettamente dipendente dalla connessione.

Per ovviare a questo inconveniente è stato necessario modificare uno degli script di gestione di Xen, in modo da non permettere che l'interfaccia fisica venga rinominata, causando la caduta della connessione[2].

Lo script in questione è `networkbridge` e la porzione di codice da modificare corrisponde alla funzione `op_start()`:

```
  op_start () {
[...]
create_bridge ${bridge}
if link_exists "$vdev"; then
        mac=‘ip link show ${netdev} | grep ’link\/ether’ |
 sed -e ’s/.*ether \(..:..:..:..:..:..\).*/\1/’‘
        preiftransfer ${netdev}
        transfer_addrs ${netdev} ${vdev}
        if ! ifdown ${netdev}; then
                # If ifdown fails, remember the IP details.
                get_ip_info ${netdev}
                ip link set ${netdev} down
                ip addr flush ${netdev}
        fi
        ip link set ${netdev} name ${pdev}
        ip link set ${vdev} name ${netdev}

        setup_bridge_port ${pdev}
        setup_bridge_port ${vif0}
        ip link set ${netdev} addr ${mac} arp on

        ip link set ${bridge} up
        add_to_bridge  ${bridge} ${vif0}
        add_to_bridge2 ${bridge} ${pdev}
        do_ifup ${netdev}

        if ! ifdown ${pdev}; then
                # If ifdown fails, remember the IP details.
                get_ip_info ${pdev}
                ip link set ${pdev} down
                ip addr flush ${pdev}
        fi

else
        ip link set ${bridge} arp on
        ip link set ${bridge} multicast on
```

[2]Nelle precedenti versioni di Xen, le interfacce di rete erano gestite senza rinominarle.

```
        # old style without ${vdev}
        transfer_addrs  ${netdev} ${bridge}
        transfer_routes ${netdev} ${bridge}
        # Attach the real interface to the bridge.
        add_to_bridge ${bridge} ${netdev}
        ip addr flush ${netdev}
fi
[...]
```

È sufficente modificare lo script in modo che la condizione dell'"if" sia sempre falsa, ovvero che venga sempre eseguito il ramo "else":

```
[...]
create_bridge ${bridge}
        ip link set ${bridge} arp on
        ip link set ${bridge} multicast on
        # old style without ${vdev}
        transfer_addrs  ${netdev} ${bridge}
        transfer_routes ${netdev} ${bridge}
        # Attach the real interface to the bridge.
        add_to_bridge ${bridge} ${netdev}
        ip addr flush ${netdev}
[...]
```

La modifica impedisce a *xend* di trasferire l'indirizzo IP dall'interfaccia di rete fisica a quella virtuale e quindi di preservare la connessione con il server NFS che fornisce il filesystem di root.

A.8 Schema della Sequenza di Boot

L'intero processo di boot per una generica workstation nel laboratorio è riassunto nella figura.

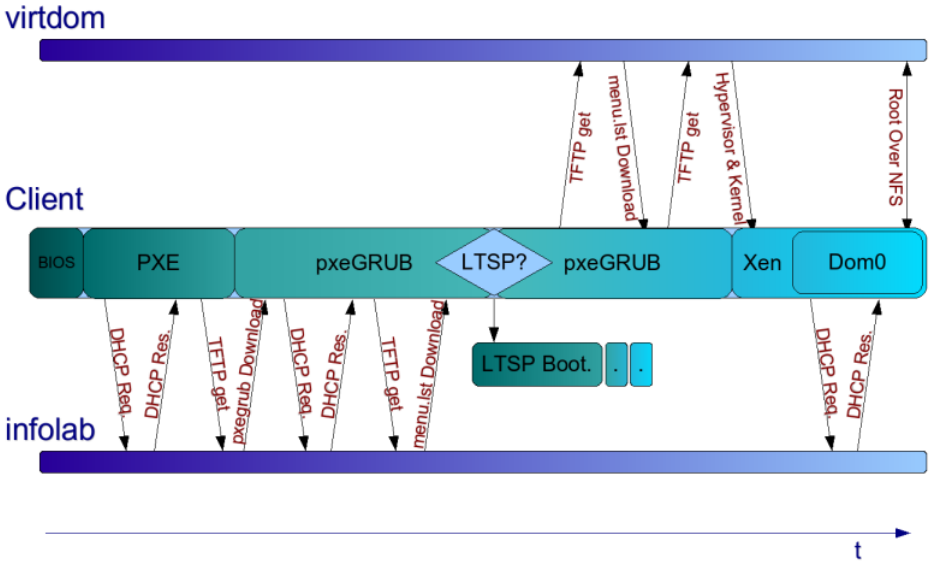


Figura A.1: Schema riassuntivo della sequenza di boot delle workstation.

Il rombo in figura rappresenta la possibilità di scelta offerta dal menu di GRUB di eseguire il normale ambiente LTSP ad uso didattico oppure impiegare la macchina come Dom0. Tale scelta, oltre che da ogni singola macchina, può essere fatta in maniera centralizzata da parte dell'amministratore.

Appendice B

Sorgenti del Prototipo

In questa appendice sono raccolti i file sorgenti del prototipo.

B.1 Manager

B.1.1 manager.py

```
#!/usr/bin/env python
#
# TODO: Tutto!
#       Comunicazione con spread. Fatto! (da ampliare)
#       Gestione gruppi. Fatto!
#       Thread ascolto separato Fatto!
#       Thread comunicazione separato. Fatto!
#       Strutture Dati client Fatto!
#       Gestione della configurazione. Fatto!
#       Migliore gestione dei Thread Fatto!
#       Decisione.
#       Interfaccia.
#       Interfaccia Scheduler.
#
# forse e' il caso di spostare le classi in file separati...

import time
import spread
from common.constants import *
from common.threadedd import ThreadedDaemon
from manager.sender import Sender, QueuedSender
from manager.listener import Listener
from manager.mconfh import MConfigHandler

class ClientDomain:
    """Client abstraction"""

    def __init__(self, name, groups = [], properties = None):
        self.name = name
        self.groups = groups
        self.active_timestamp = 0

        if properties:
            self.properties = properties
```

```
        else:
            self.properties = {}


    def __setitem__(self, label, value):
        self.properties[label] = value


    def __delitem__(self, label):
        del self.properties[label]


    def __getitem__(self, label):
        return self.properties[label]


    def __repr__(self):
        repr = '< ClientDomain: name: ' + str(self.name) + ' groups: '
        repr += str(self.groups) + 'properties: ' +  str(self.properties) + '>'
        return repr


    def keys(self):
        return self.properties.keys()


    def has_key(self, key):
        return self.properties.has_key(key)


    def join(self, group):
        if group not in self.groups:
            self.groups.append(group)


    def leave(self, group):
        try:
            self.groups.remove(group)
        except ValueError:
            # FIXME:
            # Since this exception is raised at each client rejoin after manager
            # goes down and is restarted, try to understand WHEN and WHY.
            #
            # WHEN:
            #   listener.Listener.run_body +22
            #   Manager.clients_leaves +5
            # WHY:
            #   A membership message is received for each group the client
            #   leaves... the first message causes no exception because the
            #   client is not in manager.clients list so the check avoids it.
            #   Subsequent messages do cause the exception because there is no
            #   check for exsistence of the group the client is leaving in the
            #   client's group list.
            #
            # Since checking group existence needs traversing a list (very
            # short, btw) it's good to keep the exception rising, and signal
            # the exception as a not dangerous one.
            # To do this in the cleanest way we need a proper diagnostic/verbose
            # system.
            #
            print "***WARN [Manager]: " + self.name + " already wasn't in group: " + group
```

```
    def set_active_timestamp(self):
        self.active_timestamp = time.time()

    def dump_properties(self):
        """print the properties dict to stdout"""
        for p in self.properties.keys():
            print '-' * 78
            print p + ':'
            print self.properties[p]
            print '-' * 78



class Manager:
    """Manager main class"""

    def __init__(self, conf_file='./manager.conf'):
        self.config = MConfigHandler(conf_file)
        self.triggers = self.config.get_triggers()
        self.triggers.compile_trigger()

        self.connection = None
        self.listener = None
        self.sender = None
        self.trigwatch = None

        # groups initialization
        # self.groups is a nested dictionary
        # each groups element will contain the reference to connected clients.
        # self.groups = {'group1':{'client_name1': <ClientDomain instance>,
        #                          'client_name2': <ClientDomain instance>,
        #                          ...
        #                          'client_nameN': <ClientDomain instance>},
        #                'group2':{...},
        #                'groupN':{...} }
        self.groups = {}
        for g in self.config.groups:
            self.groups[g] = {}

        self.clients = {}
        self.connect()

    def connect(self, membership = 1):
        """do the spread connection, join groups, init listener and sender on
            spread network and send messages to all clients to reconnect.
        """
        self.connection = spread.connect(str(self.config.spread_port) + '@' +
                self.config.spread_daemon, self.config.local_name, 0,
                int(membership))

        self.listener = Listener(self)

        if self.config.queued_sending:
            self.sender = QueuedSender(self,
                     self.config.queued_sending_interval)
            self.sender.begin()
        else:
            self.sender = Sender(self)

        self.trigwatch = TriggerWatch(self, self.config.triggers_eval_interval)

        #dict initialization to be used as globals by Triggers instance.
```

```
        triggers_dict = { 'manager':self,
                'config':self.config,
                'listener':self.listener,
                'sender':self.sender,
                'clients':self.clients,
                'groups':self.groups,
                'triggers':self.triggers}

        self.triggers.set_globals(triggers_dict)


        self.listener.begin()
        #we dont care about what clients do between master starts to listen
        #to messages and masters tell them to reconnect.
        for g in self.groups.keys():
            self.connection.join(g)
            #at each master reconnection clients reconnect too in order to
            #notify their presence to master.
            self.sender.send_ctrl_rjoin(g)
        self.trigwatch.begin()


    def client_joins(self, client_name, group):
        """add a client to the list, or add a group to its groups list
           the client is then added to the manager groups list"""
        if  client_name == '#' + self.config.local_name + '#' + \
                self.config.spread_daemon:
            return
        if not self.clients.has_key(client_name):
            self.clients[client_name] = ClientDomain(client_name, [group])
            #send message to query properties list.
            self.sender.send_get_proplist(client_name)
        else:
            self.clients[client_name].join(group)

        if not self.groups[group].has_key(client_name):
            self.groups[group][client_name] = self.clients[client_name]

        triggers_dict={'client_name':client_name,
                       'group_name':group,
                       'property':None,
                       'reason':spread.CAUSED_BY_JOIN}
        self.triggers.check_ouc_triggers(_locals=triggers_dict)


    def client_leaves(self, client_name, group):
        """remove group from client's groups list"""
        if  client_name == '#' + self.config.local_name + '#' + \
                self.config.spread_daemon:
            return
        if self.clients.has_key(client_name):
            self.clients[client_name].leave(group)
            if len(self.clients[client_name].groups) == 0:
                self.client_disconnects(client_name)

        if self.groups[group].has_key(client_name):
            del self.groups[group][client_name]

        triggers_dict={'client_name':client_name,
                       'group_name':group,
                       'property':None,
                       'reason':spread.CAUSED_BY_LEAVE}
```

```
        self.triggers.check_ouc_triggers(_locals=triggers_dict)


    def client_disconnects(self, client_name):
        """remove client on disconnect"""
        if  client_name == '#' + self.config.local_name + '#' + \
                self.config.spread_daemon:
            return
        if self.clients.has_key(client_name):
            del self.clients[client_name]

        for g in self.groups.values():
            print g
            if g.has_key(client_name):
                del g[client_name]

        triggers_dict={'client_name':client_name,
                       'group':None,
                       'property':None,
                       'reason':spread.CAUSED_BY_DISCONNECT}
        self.triggers.check_ouc_triggers(_locals=triggers_dict)


    def client_sent_property(self, client_name, prop, rtype, value):
        """set the 'value' of the 'property' of the client with name
        'client_name'"""
        try:
            if self.clients.has_key(client_name):
                if rtype == 'str':
                    tvalue = str(value)
                elif rtype == 'int':
                    tvalue = int(value)
                elif rtype == 'float':
                    tvalue = float(value)
                elif rtype == 'bool':
                    tvalue = bool(value)
                elif rtype == 'eval':
                    tvalue = eval(value)
                else:
                    tvalue = str(value)

        except ValueError, er:
            print '***WARN [Manager]: wrong type for property:', prop
            print '\t', er
            tvalue = None

        self.clients[client_name][prop] = tvalue

        triggers_dict={'client_name':client_name,
                       'group':None,
                       'property':prop,
                       'reason':spread.REGULAR_MESS}
        self.triggers.check_ouc_triggers(_locals = triggers_dict)

    def client_sent_properties(self, client_name, properties):
        """set properties labels"""
        if self.clients.has_key(client_name):
            for p in properties:
                self.clients[client_name][p] = None

    def client_active_timestamp(self, client_name):
        """set the timestamp value of the client as now."""
```

```
        if self.clients.has_key(client_name):
            self.clients[client_name].set_active_timestamp()

    def dump_clientsngroups(self):
        for c in self.clients.values():
            print 'client:', c.name
            for g in c.groups:
                print '\t', g
            print 'groups:'
            print self.manager.groups
            print '-----------'



class TriggerWatch(ThreadedDaemon):
    """Trigger evaluation and execution thread."""

    def __init__(self, manager, triggers_eval_interval = 1):
        ThreadedDaemon.__init__(self, name = 'TrigWatch',
                poll_interval = triggers_eval_interval)
        self.manager = manager
        self.triggers = manager.triggers

    def run_body(self):
        self.triggers.eval_trigger_ontick(self.tick)

if __name__ == '__main__':
    import sys

    conff = '/etc/dddaemon.d/manager.cfg'
    if len(sys.argv) > 1:
        conff = sys.argv[1]
        print conff

    mngr = Manager(conff)
    print 'Control left to threads...'

    """if listener stops there's no reason to keep the main thread running.
        and since all others threads (just sender actually) are daemonic
        they will terminate too."""
    while mngr.listener.isAlive() and mngr.trigwatch.isAlive():
            time.sleep(5)
```

B.1.2 sender.py

```
import spread
from common.constants import *
from common.threadedd import ThreadedDaemon


class Sender:
    """Offers methods to send messages through spread."""

    def __init__(self, manager):
        self.manager = manager
        self.spread_connection = manager.connection
        self.clients = manager.clients


    def send(self, destination, message, _type = 0):
        """send generic message"""
```

```
        self.spread_connection.multicast(spread.RELIABLE_MESS, str(destination),
                str(message), _type)


    def send_get_proplist(self, client_name = None):
        """send messages to get property list from client with passed name,
        if no name is passed, messages are sent to all known clients."""
        if not client_name:
            for c in self.manager.clients.keys():
                self.send(c, '', CLT_GETPROP)
        else:
            self.send(client_name, '', CLT_GETPROP)


    def send_get_property(self, client_name = None, prop_name = None,
            getorfetch = CLT_GETPROP):
        """Send messages to query property 'prop_name' from client named
        'client_name'.
        If no prop_name is passed, messages are sent to client named client_name
        to query all properties.
        If no client_name is passed, messages are sent to all known clients to
        query all properties; in this case prop_name is ignored.

        If prop name is '' (zero length string) and client_name not None it will
        query for property list; however, while there's no reason why it should
        not work, this has never been tested so don't use it!"""

        if not (getorfetch == CLT_GETPROP or getorfetch == CLT_FETCHPROP):
            """if an unexpected message code is passed CLT_GETPROP
                is assumed."""
            getorfetch = CLT_GETPROP

        if client_name:
            """get or fetch property or property list from client with
            client_name"""
            if prop_name:
                self.send(client_name, prop_name, getorfetch)
            else:
                for p in self.clients[client_name].keys():
                    self.send(client_name, p, getorfetch)
        else:
            """get all properties from all clients, prop_name is ignored."""
            for c in self.clients.values():
                for p in c.keys():
                    self.send(c.name, p, getorfetch)


    def send_ctrl_rjoin(self, group = None):
        """Send rejoin message.
        If no group specified, the message is sent to all known groups."""
        if group:
            self.send(group, CLT_CONTROL_RJOIN, CLT_CONTROL)
        else:
            for g in self.manager.groups.keys():
                self.send(group, CLT_CONTROL_RJOIN, CLT_CONTROL)


    def send_ctrl_rcon(self, destination = None):
        """Send reconnect message.
        If no destination specified, the message is sent to all known clients.
        destination can be a group name or a spread name."""
        if destination:
```

```
            self.send(destination, CLT_CONTROL_RCON, CLT_CONTROL)
        else:
            for c in self.clients.keys():
                self.send(c, CLT_CONTROL_RCON, CLT_CONTROL)


    def send_ctrl_stop(self, destination = None):
        """Send stop message.
        If no destination specified, the message is sent to all known clients.
        destination can be a group name or a spread name."""
        if destination:
                self.send(destination, CLT_CONTROL_STOP, CLT_CONTROL)
        else:
            for c in self.clients.keys():
                self.send(c, CLT_CONTROL_STOP, CLT_CONTROL)


    ##
    ## FIXME: I metodi seguenti non sono piu' utilizzati e dovrebbero
    ##         essere rimossi.
    ##

    def run_body(self):
        """Thread main loop.
        Since introduction of TriggerWatch this is no longer needed.
        Sending periodical messages is done by specifing triggers."""

    def query_properties(self, client_name):
        """send message to get properties list from client."""
        self.send(client_name, '', CLT_GETPROP)


    def get_property(self, client_name, property):
        """send message to fetch property value from client."""
        self.send(client_name, property, CLT_GETPROP)


    def get_properties(self, client_name):
        """send message to receive all properties' values contained in
        properties list."""
        for p in self.clients[client_name].properties:
            self.get_property(client_name, p)


    def get_clients_properties(self):
        """cycle trough clients and get all their properties."""
        for c in self.clients.keys():
           self.get_properties(c)


    def fetch_property(self, client_name, property):
        """send message to fetch property value from client"""
        """properties will be freshly fectched before being sent"""
        self.send(self.clients[client_name], property, CLT_FETCHPROP)


    def fetch_properties(self):
        """send message to receive all properties' values contained in
        properties list."""
        """properties will be freshly fectched before being sent."""
        for p in self.clients[client_name].properties:
            self.fetch_property(client_name, p)
```

```
class QueuedSender(Sender, ThreadedDaemon):
    """Same as Sender but with a queue for messages. The messages are queued
    then sent, every 'poll_interval' seconds.
    An high sending ratio (low poll_interval) will cause high cpu usage."""

    _poll_interval = 0.01

    def __init__(self, manager, poll_interval = _poll_interval):
        ThreadedDaemon.__init__(self, name = 'Sender', poll_interval =
                poll_interval)
        Sender.__init__(self, manager)
        self.messages_queue = []


    def send(self, destination, message, _type):
        """Override normal message sending. Messages got queued"""
        self.enqueue_message(destination, message, _type)


    def spread_send(self, message):
        """this is Sender.send(self, destination, message, _type)."""
        self.spread_connection.multicast(spread.RELIABLE_MESS, message[0],
                 message[1], message[2])


    def enqueue_message(self, destination, message, _type):
        """add a message to the queue to be sent."""
        self.messages_queue.append((destination, message, _type))


    def dequeue_message(self):
        """extracts the first message from queue and sends it"""
        return self.messages_queue.pop(0)


    def got_message(self):
        return (len(self.messages_queue) > 0)


    def run_body(self):
        """main thread loop.
        It just sends get property messages to all clients every
        'query_interval' seconds."""
        if self.got_message():
            mess = self.dequeue_message()
           # print mess
            self.spread_send(mess)
```

B.1.3 listener.py

```
import time
import spread
from common.threadedd import ThreadedDaemon
from common.constants import *


class Listener(ThreadedDaemon):
    """Messages listener on the spread network."""
```

```
def __init__(self, manager):
    ThreadedDaemon.__init__(self, name = 'Listener', poll_interval = 0)
    self.manager = manager
    self.spread_connection = manager.connection


def run_body(self):

    while self.spread_connection.poll() == 0 and self.running:
        time.sleep(0.1)
    smsg = self.spread_connection.receive()

    if type(smsg) == spread.MembershipMsgType:
        #print (smsg.extra, smsg.group, smsg.group_id, smsg.members,
        #        smsg.reason)
        if smsg.reason == spread.CAUSED_BY_JOIN:
            """a client joined a known group"""
            print smsg.extra[0], 'joined the group:', smsg.group
            self.manager.client_joins(smsg.extra[0], smsg.group)

        if smsg.reason == spread.CAUSED_BY_DISCONNECT:
            """a client disconnects from the network"""
            print smsg.extra[0], 'disconnected from spread network.'
            self.manager.client_disconnects(smsg.extra[0])

        if smsg.reason == spread.CAUSED_BY_LEAVE:
            """a client leaves a group, this is not expected."""
            print smsg.extra[0], 'left the', smsg.group, 'group'
            self.manager.client_leaves(smsg.extra[0], smsg.group)

        #set the timestamp of the client to the last received message time.
        self.manager.client_active_timestamp(smsg.extra[0])

    if type(smsg) == spread.RegularMsgType:
        #print (smsg.endian, smsg.groups, smsg.message, smsg.msg_type,
        #        smsg.sender)
        verbose_str = smsg.sender + ' '
        t = smsg.msg_type
        if t >= 0:
            isknown = '(unkown client)'
            if self.manager.clients.has_key(smsg.sender):
                isknown = '(known client)'
                #print self.manager.clients[smsg.sender]
            verbose_str += isknown + ' '

            if t == CLB_PROPVAL:
                #FIXME: if a malformed message is recivied, will cause the
                # manager to throw and exception and exit.
                prop, rtype, value = smsg.message.split("#",2)
                self.manager.client_sent_property(smsg.sender, prop, rtype, value)
                verbose_str += 'sent property: ' + prop

            elif t == CLB_PROPLIST:
                #FIXME: same as previous if...
                self.manager.client_sent_properties(smsg.sender,
                        smsg.message.split(","))
                verbose_str += 'sent properties list.'

            elif t == CLB_EXECRET:
                #FIXME: same as previous if...
                exe, param, res = smsg.message.split("#",2)
```

```
                verbose_str +=  'sent execution result:\n'
                verbose_str += '\tcommand: ' +  exe + ' ' + param + '\n'
                verbose_str += '\tresult:\n'
                verbose_str += res

            elif t == CLB_EXECLIST:
                verbose_str += 'sent executables list.'

            elif t == CLB_CTRLRJOIN:
                verbose_str += 'rejoined group ' + smsg.message

            elif t == CLB_CTRLRCON:
                verbose_str += 'reconnected with success. '

            #control messages; to be ignored by manager... as we are.
            elif t == CLT_CONTROL:
                if smsg.message == CLT_CONTROL_RCON:
                   verbose_str += 'sent reconnect message, ignored'
                if smsg.message == CLT_CONTROL_STOP:
                    verbose_str += 'sent stop message, ignored'

            else:
                verbose_str += 'sent UNKOWN message ' + str(t) + ' '

        else:
            #TODO: error messages handling
            verbose_str += 'sent ERROR message. ' + str(t) + \
                           ' (ERROR handling is not implemented yet!)'

        #set the timestamp of the client to the last received message time.
        self.manager.client_active_timestamp(smsg.sender)

        #FIXME: This will be removed!
        if smsg.message == 'stopmanager':
            verbose_str += 'sent stop message...'
            print verbose_str
            print 'will return in', self.manager.config.query_interval, \
                    'seconds max...'
            self.stopThreads()

        print verbose_str
```

B.1.4 mconfh.py

```
#!/usr/bin/env python

if __name__ == '__main__':
    from config import ConfigFileParser
else:
    from common.config import ConfigFileParser

class MConfigHandler(ConfigFileParser):

    config_toparse = {'spread_daemon':'str',
                              'spread_port':'int',
                              'local_name':'str',
                              'groups':'strlist',
                              'triggers_eval_interval':'float',
                              'queued_sending':'bool',
                              'queued_sending_interval':'float',
                              'user_triggers':'strlist'
```

```
                                }

    def __init__(self, config_file, config_toparse = config_toparse):
        ConfigFileParser.__init__(self, config_file, config_toparse)
        self.read(self.user_triggers)

    def parse_trigger(self, label):
        """read trigger definition from config file in trigger dict."""
        hdr = ' Found trigger ' + label + ' '
        hlen = 78 - len(hdr)

        print '[' + '-' * (hlen / 2) + hdr + '-' * (hlen / 2) + ']'

        expr = self.get(label, 'expr')[1:-1].replace('----','    ').strip()
        print 'Expression:', expr

        cmd = self.get(label, 'cmd')[1:-1].replace('----','    ').strip()
        if cmd.startswith('file:'):
            fname = cmd.split(':',1)
            print fname[0], fname[1]
            f = open(fname[1])
            cmd=f.read()
            f.close()

        print 'Command:\n'
        print cmd

        hdr = '-'
        hlen = 78 - len(hdr)

        print '\n[' + '-' * (hlen / 2) + hdr + '-' * (hlen / 2) + ']\n'

        ouc = self.getboolean(label, 'on_update_check')
        ticks = self.getint(label, 'ticks')
        trg = [expr, cmd, ticks, ouc]
        return trg

    def parse_triggers(self):
        """parse all triggers definitions"""
        sections = self.sections()
        triggers = {}
        for t in sections:
            if not t == self.config_section:
                triggers[t] = self.parse_trigger(t)
        return triggers

    def get_triggers(self, _globals = None, _locals = None):
        """returns the trigger object rapresentation parsed from config
        file."""
        return Triggers(self.parse_triggers(), _globals, _locals)

class Triggers:
    """Write something here. This class is basically the same as
    client.cconfig.Properties. It could be useful to use some inheritance."""

    def __init__(self, triggers = None, _globals = None, _locals = None):
        if triggers:
            self.triggers = triggers
            self.ouc_triggers = self.get_on_update_triggers()
        else:
            self.triggers = {}
```

```
        self._globals = _globals
        self._locals = _locals
        self.code_objects = {}

    def __setitem__(self, label, value):
        self.code_objects[label] = value

    def __delitem__(self, label):
        del self.code_objects[label]

    def __getitem__(self, label):
        return self.code_objects[label]
        #if label not in self.values.keys(): return None ???

    def __repr__(self):
        return str(self.code_objects)

    def keys(self):
        return self.code_objects.keys()

    def has_key(self, key):
        return self.code_objects.has_key(key)

    def get_on_update_triggers(self):
        """returns the list of triggers with on_update_check field set true"""
        trgs = {}
        for t in self.triggers.keys():
            if self.triggers[t][3]:
                trgs[t] = self.triggers[t]
                print "OUC_Trigger:[", t ,"]:"
        return trgs

    def compile_trigger(self, label = None):
        """Compile trigger 'expr' string and saves the compiled code object.
        if label not set all trigger are compiled."""
        #FIXME: no check is done on label existence.
        if label:
            self.code_objects[label] = (compile(self.triggers[label][0], label,
                   'eval'), compile(self.triggers[label][1], label, 'exec'))
        else:
            for t in self.triggers.keys():
                self.compile_trigger(t)

    def set_globals(self, _globals):
        self._globals = _globals

    def set_locals(self, _locals):
        self._locals = _locals

    def eval_trigger(self, label, _globals = None, _locals = None,
        execute = False):
        """Trigger evaluation. The trigger must have been compiled first. If
        execute is False (default), the trigger is just evaluated and the value
        is returned.
        Otherwise, if trigger evaluation is True, its command code will be
        executed and the value returned.
        Globals and locals can be used to restrict access to names.
        """

        #TODO: Define behaviour if trigger expression contains undefined names:
        #           - Raise exception?
        #           - Catch exception and return 'False'?
```

```
        if not _globals:
            _globals = self._globals

        if not _locals:
            _locals = self._locals

        try:
            t_value = eval(self.code_objects[label][0], _globals, _locals)
        except Exception, ex:
            print '*** [Triggers] Exception while evaluating', label, ':'
            print '\t', ex
            return

        if execute and t_value:
            print 'Executing triggered cmd...', label, ':'
            try:
                exec(self.code_objects[label][1], _globals, _locals)
            except Exception, ex:
                print '***WARN [Triggers] Exception while executing', label, ':'
                print '\t', ex
                return

        return t_value

    def check_ouc_triggers(self, _globals = None, _locals = None):
        for t in self.ouc_triggers.keys():
            self.eval_trigger(t, _globals, _locals, True)

    def eval_trigger_ontick(self, cycle, _globals = None, _locals = None):
        """Evaluation of all triggers according to tick value.
        Cycle is the ThreadedDaemon cycle count.
        Triggers are evaluated according to following rules:
            if t tick is < 0: trigger will never be evaluated.
            if t. t. > 0: trigger will be fetched each t.t. cycle.
            if t. t. = 0: trigger will be fetched once."""
        for t in self.triggers.keys():
            tick = self.triggers[t][2]
            if tick < 0:
                continue
            elif tick == 0:
                #print 'eval just once', cycle, self.triggers[t]
                self.eval_trigger(t, _globals, _locals, True)
                self.triggers[t][2] = -1
                continue
            if (cycle % tick) == 0:
                #print 'eval', cycle, self.triggers[t]
                self.eval_trigger(t, _globals, _locals, True)

if __name__ == '__main__':

    #FIXME:Just some tests, to be removed.
    c = MConfigHandler('../test2.cfg')
    print '\n*** Going to parse triggers...\n'
    trigs = c.get_triggers()
    trigs.compile_trigger()
    print '*** Evaluating triggers...\n'
    for t in trigs.keys():
        ev = trigs.eval_trigger(t, _globals = {'x':9, 'y':90})
        print 'Trigger', t, 'is', ev

    for t in trigs.keys():
```

```
        ev = trigs.eval_trigger(t, _globals = {'x':9, 'y':90}, execute = True )
```

B.2 Client

B.2.1 client.py

```
#!/usr/bin/env python
#
#
#
#

"""Client definition. Each client has two deamons: Reader and Interface
Reader fetches information from the local host, Interface listen to messages
and triggers reactions to them."""

import spread
import socket
import time
from common.threadedd import ThreadedDaemon
from common.constants import *
from client.runner import ExeHandler
from client.cconfh import CConfigHandler


class ReaderDaemon(ThreadedDaemon):
    """Fecthes properties as described by configuration file."""

    def __init__(self, client):
        ThreadedDaemon.__init__(self, name = 'Reader',
                poll_interval = client.config.fetch_interval)
        self.executor = client.executor
        self.config = client.config
        self.properties = client.properties

    def run_body(self):
        """it just fetches all properties according to their settings."""
        self.properties.fetch_properties_ontick(self.executor, self.tick)

class IfaceDaemon(ThreadedDaemon):
    """Interface with the internal ReaderDeamon and messaging system."""

    def __init__(self, client):
        ThreadedDaemon.__init__(self, name = 'Iface', poll_interval = 0)
        self.config = client.config
        self.spread_connection = client.spread_connection
        self.executor = client.executor
        self.properties = client.properties
        self.client = client

    #write status response
    def write_response(self, message, type):
        m = '#' + self.client.config.manager_name + '#' + \
             self.client.config.spread_daemon
        print 'to: ', m, 'type:', type
        print '\t',message
        print '----------'
        self.spread_connection.multicast(spread.RELIABLE_MESS, m, str(message),
                type)
```

```
def run_body(self):

    #['endian', 'groups', 'message', 'msg_type', 'sender']

    while self.spread_connection.poll() == 0 and self.running:
        time.sleep(0.1)
    msg = self.spread_connection.receive()

    print 'from:', msg.sender, 'to', msg.groups, 'type', msg.msg_type
    print '\t', msg.message

    #TODO
    #if message.sender != manager_name:
    #   print message ignored
    #   return

    mess = msg.message.split()

    if mess == None:
        return

    #noop | test = OK (timestamp o altro?)
    if msg.msg_type == CLT_NOOP:
        res = 'noop'
        res_t = CLB_NOOP

    #get or fetch property | test = OK
    elif msg.msg_type == CLT_FETCHPROP or msg.msg_type == CLT_GETPROP:
        if len(mess) >= 1:
            if self.properties.has_key(mess[0]):
                if (msg.msg_type == CLT_FETCHPROP):
                    print '\t' + '^' * len(mess[0]) + 'freshly fetching!'
                    self.properties.fetch_property(mess[0], self.executor)
                # self.properties.properties[mess[0]][2] is properties type.
                # '<propname>#<type>#<value>'
                res = mess[0] + '#' + self.properties.properties[mess[0]][2]
                res += '#' + str(self.properties[mess[0]])
                res_t = CLB_PROPVAL
            else:
                res = "no property named '%s'." % mess[0]
                res_t = CLB_PROPERR
        else:
            res = ''
            for k in self.properties.keys():
                res += k + ','
            res = res[:-1]
            res_t = CLB_PROPLIST

    #exec | test = OK
    elif msg.msg_type == CLT_EXECUTE:
        if len(mess) >= 1:
            params = ''
            for p in mess[1:]:
                params = params + ' ' + p
            #add some identifiers strings
            res = mess[0] + '#' + params + '#' + self.executor.execute(mess[0], params)
            res_t = CLB_EXECRET
        else:
            res = str(self.executor.exe_names())
            res_t = CLB_EXECLIST
```

```
        #ctrl | test = OK
        elif msg.msg_type == CLT_CONTROL:
            if len(mess) >= 1:

                #stop daemons.
                if mess[0] == CLT_CONTROL_STOP:
                    print 'stopping all daemons...'
                    res = ''
                    for i in self.getThreads():
                        res += str(i)
                        i.end()
                    res = 'all daemons halted.'
                    res_t = CLB_CTRLSTOP

                #reconnect to spread network.
                elif mess[0] == CLT_CONTROL_RCON:
                    self.spread_connection = self.client.connect()
                    res = 'reconnection ok.'
                    res_t = CLB_CTRLRCON
                    #what to do in case of error??

                #rejoin group.
                elif mess[0] == CLT_CONTROL_RJOIN:
                    grp = msg.groups[0]
                    self.spread_connection.leave(grp)
                    self.spread_connection.join(grp)
                    res = grp
                    res_t = CLB_CTRLRJOIN

                else:
                    res = "unkown control '%s'." % mess[0]
                    res_t = CLB_CTRLUNKN
            else:
                res = 'malformed control.'
                res_t = CLB_CTRLBAD

        #unkown message | test = OK
        else:
            res = 'unkown message type.'
            res_t = CLB_UNKNOWNTYPE

        #TODO
        #get destination from client object!!
        #self.write_response(self.config.manager_name, response)
        self.write_response(res, res_t)

class Client:
    def __init__(self, conf_file = './client.conf'):
        self.config = CConfigHandler(conf_file)
        self.executor = ExeHandler(self.config.exe_path)
        self.executor.find_exes()
        self.properties = self.config.get_properties()
        self.properties.fetch_properties(self.executor)
        self.spread_connection = None
self.iface = None
self.reader = None


    def disconnect(self):
        """disconnect from spread network"""
        if not self.spread_connection == None:
            self.iface.end()
```

```
            self.reader.end()

    del self.iface
    del self.reader

            self.spread_connection.disconnect()

    def connect(self, membership = 0):
        """do the spread connection and join groups.
        If connection exist, do a new connection"""
        self.disconnect()
        self.spread_connection = spread.connect(str(self.config.spread_port) +
                '@' + self.config.spread_daemon, self.config.local_name, 0,
                int(membership))
        for g in self.config.groups:
            self.spread_connection.join(g)

self.iface = IfaceDaemon(self)
        self.reader = ReaderDaemon(self)

self.iface.begin()
        self.reader.begin()

        return self.spread_connection

if __name__ == '__main__':
    import sys

    #TODO better arguments handling, this is a dirty hack around
    conff = '/etc/dddaemon.d/client.cfg'
    cname = None

    if len(sys.argv) > 2:
        conff = sys.argv[1]
        cname = sys.argv[2]
    elif len(sys.argv) == 2:
        conff = sys.argv[1]

    clt = Client(conff)
    if cname:
        print cname, clt.config.local_name
        clt.config.local_name = cname
    clt.connect()
    print 'Control left to threads...'

    """if iface stops there's no reason to keep the main thread running.
        and since all others threads (just reader actually) are daemonic
        they will terminate too."""
    while (clt.iface is not None) and (clt.iface.isAlive()):
        time.sleep(5)

    print 'Terminated...'
```

B.2.2 cconfh.py

```
#
# Client properties and config handler
#
```

```
import runner
from common.config import ConfigFileParser
from ConfigParser import RawConfigParser
from ConfigParser import NoSectionError
from ConfigParser import NoOptionError

class CConfigHandler(ConfigFileParser):
    """Client configuration parser and handler"""

    config_toparse = {'exe_path':'strlist',
                              'fetch_interval':'float',
                              'spread_daemon':'str',
                              'spread_port':'int',
                              'local_name':'str',
                              'manager_name':'str',
                              'groups':'strlist',
                              'user_properties':'strlist'
                               }

    def __init__(self, config_file, config_toparse = config_toparse):
        ConfigFileParser.__init__(self, config_file, config_toparse)
        self.read(self.user_properties)

    def parse_property(self, label):
        """read property definition from config file into properties dict."""
        cmd = self.get(label,'command')
        parms = self.get(label,'parameters')
        rtype = self.get(label, 'return_type')
        ticks = self.getint(label,'ticks')
        prop = [cmd, parms, rtype, ticks]
        return prop

    def parse_properties(self):
        """parse all properties"""
        props_sections = self.sections()
        props = {}
        for p in props_sections:
            if not p == self.config_section:
                props[p] = self.parse_property(p)
        return props

    def get_properties(self):
        """returns the properties object
        rapresentation parsed from config file."""
        return Properties(self.parse_properties())

class Properties:
    """Offers easy methods to fetch properties and access properties values"""

    #TODO: aggiungere una flag per ogni proprieta' che sia vera quando il
    # suo valore e' freshly fetched e venga messa a 0 quando viene inviata

    def __init__(self, properties = None):
        if properties:
            self.properties = properties
        else:
            self.properties = {}

        self.values = {}

    def __setitem__(self, label, value):
        self.values[label] = value
```

```
    def __delitem__(self, label):
        del self.values[label]

    def __getitem__(self, label):
        print label
        return self.values[label]
        #if label not in self.values.keys(): return None ???

    def keys(self):
        return self.values.keys()

    def has_key(self, key):
        return self.values.has_key(key)

    def fetch_property(self, label, exe_handler):
        """executes the script defined by property and saves the result in the
        'values' dict"""
        if self.properties.has_key(label):
            value = exe_handler.execute(self.properties[label][0],
                self.properties[label][1])
            self.values[label] = value
            return value
        else:
            return None

    def fetch_properties(self, exe_handler):
        """fetches the values for all properties"""
        for p in self.properties.keys():
            self.fetch_property(p, exe_handler)

    def fetch_properties_ontick(self, exe_handler, cycle):
        """fetches properties according to tick value. Cycle is the ReaderDaemon
        cycle count. Properties are fetched according to following rules:
            if property tick is < 0: property will never be fetched.
            if p. t. > 0: property will be fetched each p.t. cycles.
            if p. t. = 0: property will be fetched once."""
        for p in self.properties.keys():
            tick = self.properties[p][3]
            if tick < 0:
                continue
            elif tick == 0:
                self.fetch_property(p, exe_handler)
                self.properties[p][3] = -1
                continue
            if (cycle % tick) == 0:
                self.fetch_property(p, exe_handler)
```

B.2.3 runner.py

```
#!/usr/bin/env python
#
# "Esecutore generico".
#
# Ricerca tutti gli eseguibili in una lista di directory e
# permette di eseguirli.
#

import os
import commands
```

```
class ExeHandler:
    """Executables handler. Searches and provide handlers to execute files
    from the given path"""

    def __init__(self, exe_path = ['./scripts']):
        self.exe_path = exe_path
        self.exes = []

    def find_exes(self):
        """load all executables from exe_path"""
        self.exes = []
        for dir in self.exe_path:
            for file in os.listdir(dir):
                fullpath = os.path.join(dir,file)
                if file is not None and os.path.isfile(fullpath) and  \
                        os.access(fullpath, os.X_OK):
                        self.exes.append(fullpath)
        return self.exes

    def execute(self, exe, params = ' '):
        """executes the exe with params parameters and returns the output"""
        if exe is None:
            print 'No executable name provided.'
            return None
        exe_fp = self.fullpath(exe)
        if exe_fp not in self.exes:
            print 'Executable %s not found.' % exe
            return None
        return commands.getoutput(exe_fp + ' ' + params)

    def fullpath(self, exe):
        """returns the fullpath of the executable passed as parameter"""
        for ex in self.exes:
            (dir,file) = os.path.split(ex)
            if exe == file:
                return ex
        return None

    def exe_names(self):
        """returns names of all executables"""
        exe_names = []
        for exe in self.exes:
            exe_names.append(os.path.split(exe)[1])
        return exe_names

    def got_exe(self, exe):
        """true if executable is in exe_path"""
        return exe in self.exe_names()

if __name__ == '__main__':

    import getopt
    import sys

    def usage():
        print "USAGE:"
        print " %s [-h | --help] [[-p | --exe_path path1][, path2][, ...]]", \
            "[ -l | --exe_list] [command_name [arguments...]]" % sys.argv[0]
        print
        print " -h | --help                              ", \
```

```
        "Print this help and exits."
    print " -p | --exe_path path1[, path2][, ...]   ", \
        "Search for executables in the given paths."
    print " -l | --exe_list                         ",\
        "List all found executables. No command are executed",\
        "(even if specified) when given this flag."
    print " command_name [arguments]                ",\
        "Executes the command given as 'command_name' with specified",\
        "'arguments'. "
    print
    print " If no command specified [-l | --exe_list]",\
        "is assumed by default."
    print " If no [-p | --exe_path] is given current working directory",\
        "is assumed by default"


try:
    opts, args = getopt.getopt(sys.argv[1:], "hlp:", ["help", "exe_list"
        "exe_path"])
except getopt.GetoptError:
    # print help information and exit:
    usage()
    sys.exit(2)

#defaults
exe_path = ['.']
exe_list = False
if len(args) == 0 :
   exe_list = True

for opt, arg in opts:

    if opt in ("-h", "--help"):
        usage()
        sys.exit()

    #set exe_path
    if opt in ("-p", "--exe_path" ):
        exe_path = arg.split(",")

    #list all exec
    if opt in ("-l", "--exe_list"):
        exe_list = True

eh = ExeHandler(exe_path)
eh.find_exes()

if exe_list:
    print eh.exe_names()
    sys.exit()


if len(args) == 1:
    print eh.execute(args[0])
else:
    print eh.execute(args[0],args[1])

# print exer.fullpath('ascript')
```

B.3 Comuni

B.3.1 config.py

```
#
# Configuration file-parser and handler base class
#

from ConfigParser import RawConfigParser
from ConfigParser import NoSectionError
from ConfigParser import NoOptionError

class ConfigFileParser(RawConfigParser):
    """Simple Configuration File Parser"""

    config_toparse = {}

    def __init__(self, config_file, config_toparse = config_toparse,
            config_section = 'config'):
        RawConfigParser.__init__(self)
        self.config_file = config_file
        self.config_toparse = config_toparse
        self.config_section = config_section
        self.readfp(open(config_file))
        self.parse_config()

    def __setattr__(self, name, value):
         self.__dict__[name] = value

    def parse_config(self):
        """read configuration attributes."""

        for k in self.config_toparse.keys():
            t = self.config_toparse[k]
            if t == 'str':
                self.__setattr__(k, self.get(self.config_section, k))
            elif t == 'strlist':
                self.__setattr__(k, self.get(self.config_section, k).split(","))
            elif t == 'float':
                self.__setattr__(k, self.getfloat(self.config_section, k))
            elif t == 'int':
                self.__setattr__(k, self.getint(self.config_section, k))
            elif t == 'bool':
                self.__setattr__(k, self.getboolean(self.config_section, k))
```

B.3.2 constants.py

```
#
# constants
#


# messages types to client
CLT_NOOP = 0
CLT_GETPROP = 1
CLT_FETCHPROP = 2
CLT_EXECUTE = 3
CLT_CONTROL = 100
```

```
CLT_CONTROL_RCON = 'rcon'
CLT_CONTROL_STOP = 'stop'
CLT_CONTROL_RJOIN = 'rjoin'

# callback from client messages types
CLB_NOOP = 0

CLB_PROPVAL = 10
CLB_PROPLIST = 11
CLB_PROPERR = -10

CLB_EXECRET = 30
CLB_EXECLIST = 31

CLB_CTRLSTOP = 101
CLB_CTRLRCON = 102
CLB_CTRLRJOIN = 103
CLB_CTRLUNKN = -100
CLB_CTRLBAD = -101

CLB_UNKNOWNTYPE = -1000
```

B.3.3 threadedd.py

```
#
# Threaded daemon base class
#

import threading
from threading import Thread
from time import sleep

class ThreadedDaemon(Thread):
    """Generic daemon class definition. Provides methods to access all other
    threads."""

    def __init__(self, name = None, poll_interval = 5):
        Thread.__init__(self, name = name)
        self.poll_interval = poll_interval
        self.tick = 0
        self.running = False
        self.setDaemon(True)

    def begin(self):
        if not self.running:
            self.running = True
            self.tick = 0
            self.start()

    def end(self):
        if self.running:
            self.running = False

    def set_poll_interval(self, interval):
        try:
            interval = float(interval)
            if not (interval > 0 and interval < 10):
                raise ValueError
        except ValueError:
            return -1
```

```
        self.poll_interval = float(interval)
        return self.poll_interval

    def countThreads(self):
        """Returns the number of active threads, calling
        threading.activeCount()"""
        return threading.activeCount()

    def getThreads(self):
        """Returns a list of the instances of all active threads."""
        return threading.enumerate()

    def stopThreads(self):
        """Stops all instances registered with this ThreadedDaemon.
        """
        for i in self.getThreads():
            tn = i.getName()
            if tn != 'MainThread':
                print 'Going to stop' , tn
                i.end()

    def run(self):
        """Generic run cycle.
           run_body is not defined and must be defined by inheritors.
           run_body is then called at each cycle."""
        while self.running:
            self.tick += 1
            self.run_body()
            if self.poll_interval > 0:
                sleep(self.poll_interval)
```

B.3.4 sendmetrix.py

```
#!/usr/bin/env python
#
# python port di sendmetrix.c
#

from socket import socket, AF_INET, SOCK_DGRAM, IPPROTO_UDP
import time
import sys

ERR_MAN = "NAME \nsendmetric - Send a mentric to a remote host\n\nSYNTAX\n"
ERR_MAN += "sendmetric -h <host in dotted decimal> -p <UDP port number> -i "
ERR_MAN += "<metric name> -t {int,float,string} -d <value>\n"

SEPARATORE = ";"

def timestamp():
    tmp = time.localtime()
    return  "%04d%02d%02d%02d%02d%02d" % (tmp[0], tmp[1], tmp[2], \
           tmp[3], tmp[4], tmp[5])

def sendmetrix(host, port, id_, metric, value, daq_time = None, verbose = False):

    if not daq_time:
        daq_time = int(time.time())

    try:
```

```
        sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
    except Exception, ex:
        #dovrebbe essere stampato su stderr
        print 'Cannot create the socket:', ex
        sys.exit(1)

    buffer_ = "%s%s%s%s%s%s%s" % (id_, SEPARATORE, metric, SEPARATORE, value,
            SEPARATORE, daq_time)

    if (verbose):
        print 'Sending UDP:'
        print buffer_

    if  not (sock.sendto(buffer_, (host, int(port))) - len(buffer_)) == 0:
        print "Mismatch in number of sent bytes: %s" % len(buffer_)
        sys.exit(1)

    sock.close()

if __name__ == "__main__":
    import getopt

    #data_send
    host = None
    port = None
    id_ = None
    metric = None
    value = None

    verbose_flag = False

    try:
        opts, args = getopt.getopt(sys.argv[1:], "vh:p:i:t:d:",
                ["verbose", "host", "port", "id", "type", "value"])
    except getopt.GetoptError:
        sys.exit(1)

    for opt, arg in opts:

        if opt in("-v", "--verbose"):
            verbose_flag = True
            print "verbose flag is set."

        elif opt in("-h", "--host"):
            host = arg

        elif opt in("-p", "--port"):
            port = arg

        elif opt in("-i", "--id"):
            id_ = arg

        elif opt in("-t", "--type"):
            metric = arg

        elif opt in("-d", "--value"):
            value = arg

        elif opt  == "?" or opt == ":":
            print "%s" % ERR_MAN
            sys.exit(1)
        else:
```

```
            pass

    if not host or not port or not id_ or not metric:
        print "%s" % ERR_MAN
        sys.exit(1)

    sendmetrix(host, port, id_, metric, value, verbose_flag)

    sys.exit(0)
```

B.3.5 spreadutils.py

```
#!/usr/bin/env python

import socket
import spread

class SpreadConnection:
    """Provides a spread connection wrapper to send and receive messages"""

    def __init__(self, spread_daemon = 'localhost',
            port = spread.DEFAULT_SPREAD_PORT, groups = ['default'],
            local_name = None):
        self.spread_daemon = spread_daemon
        self.port = port
        self.groups = groups
        self.local_name = local_name

        if not local_name:
            self.local_name = socket.gethostbyaddr(socket.gethostname())[0]

    def connect(self, membership = 0):
        """do the spread connection"""
        self.connection = spread.connect(str(self.port) + '@' +
                self.spread_daemon, self.local_name, 0, int(membership))
        for g in self.groups:
            self.connection.join(g)

    def receive(self):
        """blocking if no messages"""
        return self.connection.receive()

    def write(self, destination, message):
        """send a message to destination"""
        self.connection.multicast(spread.RELIABLE_MESS, str(destination),
                str(message))

class MasterConnection(SpreadConnection):
    def __init__(self, spread_daemon = "localhost",
            port = spread.DEFAULT_SPREAD_PORT, groups = ['default'],
            local_name = None):
        SpreadConnection.__init__(self, spread_daemon, port, groups, local_name)
        self.connect(1)

class SlaveConnection(SpreadConnection):
    """spread connection for slaves: A slave can send only to the manager and
    doesnt receive membership messages"""

    def __init__(self, spread_daemon = "localhost",
            port = spread.DEFAULT_SPREAD_PORT, groups = ['default'],
```

```
            local_name = None, manager_name = 'manager'):
        """
        spread_daemon = hostname of the running spread daemon
        port = port of the spread daemon
        group_name = spread group to connect to (default: 'default')
        local_name = name with which the local daemon will be know in the group
        (default: local host name) manager_name = name of the manager of the
        group(default: 'manager')
        """
        SpreadConnection.__init__(self, spread_daemon, port, groups, local_name)
        self.manager_name = manager_name
        self.connect()

    def readline(self):
        return self.receive().message

    def write(self, message):
        SpreadConnection.write(self, "#" + self.manager_name + "#"
                + self.spread_daemon, message)
```

Bibliografia

[1] P.R. Barham, B. Dragovic, and K.A. Fraser. Xen 2002. 2003.

[2] B. Quetier, V. Neri, and F. Cappello. Selecting A Virtualization System For Grid/P2P Large Scale Emulation. *Proc of the Workshop on Experimental Grid testbeds for the assessment of largescale distributed applications and tools (EXPGRID06), Paris, France, 19-23 June*, 2006.

[3] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, 2006.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[5] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J.N. Matthews. Xen and the art of repeated research. *Proceedings of the Usenix annual technical conference, Freenix track*, pages 135–144, 2004.

[6] R. Buyya. *High Performance Cluster Computing: Architectures and Systems, Volume I.* Prentice Hall, 1999.

[7] M. Baker et al. Cluster Computing White Paper. *Arxiv preprint cs.DC/0004014*, 2000.

[8] G.F. Pfister. *In search of clusters*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1998.

[9] BC Brock, GD Carpenter, E. Chiprout, ME Dean, PL De Backer, EN Elnozahy, H. Franke, ME Giampapa, D. Glasco, JL Peterson, et al. Experience with building a commodity intel-based ccNUMA system. *IBM J. RES. DEV*, 45(2):207–227, 2001.

[10] H.W. Meuer. The TOP500 Project: Looking Back over 15 Years of Supercomputing Ex-perience. 2008.

[11] J.J. Dongarra, H.W. Meuer, E. Strohmaier, and altri. TOP500 Supercomputer Sites. *URL http://www. top500. org/.(updated every 6 months)*, 2008.

[12] F. Cantini. Studio delle prestazioni di un batch system basato su torque/maui, 2007.

[13] I. Foster. What is the Grid? A Three Point Checklist. *Grid Today*, 1(6):22–25, 2002.

[14] I. Foster. The Grid: A new infrastructure for 21st century science. *Physics Today*, 55(2):42–47, 2002.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid.

[16] M. Baker, R. Buyya, and D. Laforenza. Grids and Grid technologies for wide-area distributed computing.

[17] I. Neri. Installazione, configurazione e monitoraggio di un sito INFN-Grid., 2005.

[18] Lcg project overview.

[19] G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[20] M. Tim Jones. Virtual linux an overview of virtualization methods, architectures, and implementations. 2006.

[21] S.N.T. Chiueh. A Survey on Virtualization Technologies.

[22] R. Rose. Survey of System Virtualization Techniques. 2004.

[23] J.S. Robin and C.E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. *Proceedings of the 9th conference on USENIX Security Symposium-Volume 9 table of contents*, pages 10–10, 2000.

[24] M.F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. *ACM SIGOPS Operating Systems Review*, 40(2):8–11, 2006.

[25] W. Huang, J. Liu, B. Abali, and D.K. Panda. A Case for High Performance Computing with Virtual Machines.

[26] H.K.F. Bjerke. *HPC Virtualization with Xen on Itanium.* PhD thesis, Master thesis, Norwegian University of Science and Technology (NTNU), July 2005.

[27] A. L. Lopez. Uso di macchine virtuali (XEN) per garantire servizi di Grid, 2006.

[28] *Xen 3.0 Users Manual*, 2008.

[29] *Xen 3.0 Interface Manual for x86*, 2008.