UNIVERSITA' DEGLI STUDI DI PISA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dipartimento di Informatica

Meccanismi RPC/RMI per calcolo Parallelo e Distribuito

Relatore: Prof. Francesco Romani

Tesi di Laurea di:
Cecchi Massimo, matricola 160638

Anno Accademico 2005-2006

Indice

1 Introduzione **1**

2 Remote Procedure Call **4**

2.1 *RPC* in *C* 8

2.1.1 La semantica 8

2.1.2 Implementazione del *RPC* 11

2.2 RPC in Java 19

2.2.1 Differenze tra RPC ed RMI 20

2.3 RPC con CORBA 23

3 Implementazione del prototipo MasterSlave **33**

3.1 Definizione del modello Master/Slave 33

3.2 Architettura Astratta 34

3.3 Implementazione 37

3.3.1 Descrizione del prototipo 38

3.3.2 Il modulo Master 39

4 Esperimenti **42**

4.1 Caratteristiche di efficienza e performance del prototipo 42

4.2 Descrizione dell'ambiente di esecuzione 43

4.3 Il prototipo FARM 43

4.4 Descrizione dello stato dell'ambiente di programmazione . . . 44

4.5 Risultati 45

5 Conclusioni **55**

Bibliografia **58**

A AppendiceA **60**

A.1 Introduzione 60

A.2 Codice del prototipo oggetto della tesi: 60

A.2.1 array.x 60

A.2.2 array_clnt.c 62

A.2.3 array_svc.c 64

A.2.4 array_xdr.c 67

A.2.5 error.c 69

A.2.6 file.c 70

A.2.7 mainMaster.c 75

A.2.8 manager.c 76

A.2.9 remoto.c 83

A.2.10 slave.c 84

A.2.11 task.c 85

Capitolo 1

Introduzione

La Remote Procedure Call o *RPC* [6, 13, 14] è un'estensione, in ambiente distribuito, del meccanismo di chiamata di procedura tradizionale previsto dai linguaggi di programmazione di tipo imperativo. Il concetto di RPC nasce alla fine degli anni '70. La prima implementazione popolare di *RPC* (su UNIX) è stata la *RPC* di Sun Microsystem, la quale risale ai primi anni '80. Il modello *RPC* estende la programmazione sequenziale al contesto distribuito utilizzando il paradigma client/server [13]. Il programmatore invoca la procedura remota con le stesse modalità con cui invoca la procedura locale, senza dover necessariamente tener conto dei dettagli delle diverse piattaforme su cui gira la parte dell'applicazione chiamante e la parte dell'applicazione chiamata. Il meccanismo *RPC* si basa sull'utilizzo di procedure di interfaccia (dette stub) che implementano, in maniera trasparente al programmatore, la comunicazione client/server. In particolare abbiamo che lo stub del client preleva i parametri, dalla procedura che invoca la chiamata remota, li impacchetta in un formato indipendente alla macchina e li invia al server. Il server preleva i parametri dal messaggio giunto dal client, esegue la chiamata della procedura (locale), impacchetta il risultato in un formato indipendente alla macchina e lo invia al client. Il client che era in attesa del messaggio preleva il risultato e continua la sua esecuzione. Le operazione svolte dagli stub per l'inserimento e l'estrazione dei parametri della chiamata di proce-

dura prendono il nome di marshalling/unmarshalling. Nei sistemi *RPC* la comunicazione fra processi è intrinsecamente di tipo sincrono-bloccante. Il processo chiamante rimane sospeso fino al completamento della procedura remota. Per poter effettuare la chiamata di procedura remota sono necessari l'indirizzo dell'host su cui risiede il processo server, il nome della procedura da invocare in remoto. Inoltre RPC implementano la trasparenza della posizione. Per questo esiste un processo *binder* che gestisce i servizi disponibili su un determinato nodo. Una procedura RPC si può registrare o può verificare se un servizio è a disposizione nel *binder* [6, 14]. I File di stub e skeleton vengono generati in maniera automatica (esempio in RPC di Sun il compilatore `rpcgen` genera i suddetti file). La specifica dei generatori di codice è data dall' IDL [1] [1, 15]. L'IDL serve oltre a generare il codice di stub a collegare le procedure applicative a quelle dello strato Middleware.

In questa tesi viene proposto una rassegna su varie istanze del concetto di RPC, in particolare su RPC di Sun Microsystem [6, 13, 14], su JavaRMI [9, 16, 17] e un'introduzione a CORBA [1, 8, 15]. Viene inoltre introdotto il concetto di Master/Slave, le differenze tra questo modello e il modello Client/Server per riuscire cosi' a progettare l'architettura astratta del prototipo implementato in questa tesi. Il prototipo implementato in questa tesi è un prototipo di ambiente di programmazione parallelo strutturato basato su RPC e con implementazione secondo il modello Master/Slave. Mediante tale prototipo e tramite il monitoraggio del tempo di completamento in base al grado di parallelismo si è calcolata l'efficienza (quindi scalabilità) dell'esecuzione in remoto di un'applicazione generica di tipo task farm [18]. Dopo un attento studio dei vari linguaggi di programmazione RPC è stato utilizzato l'ambiente di programmazione C [11, 12] e RPC di sun. È stata fatta questa scelta di implementazione soprattutto per avere alto grado di efficienza (scalabilità).

La tesi è così organizzata: nel Capitolo 2 viene introdotta la definizione di RPC, vengono descritti alcuni ambienti di programmazione di RPC, in

[1] IDL: dall'acronimo Interface Definition Language

particolare nell'ambiente RPC, javaRMI di Sun Mycrosystem C e CORBA. Nel Capitolo 3 viene data la definizione del Modello Master/Slave. Inoltre è stata definita l'architettura astratta, l'implementazione e la descrizione del prototipo di ambiente per il calcolo parallelo distribuito implementato secondo la logica Master/Slave oggetto della tesi. Nel Capitolo 4 vengono proposti degli esperimenti effettuati sul prototipo per vederne le caratteristiche di efficienza e performance, in particolare viene descritto l'ambiente di esecuzione, il tipo di parallelismo sfruttato, i risultati ottenuti ed alcuni grafici per capire meglio i risultati ottenuti. Nel Capitolo 5 vengono ripetuti i risultati e le conclusioni. Infine, in AppendiceA, si riporta il codice utilizzato per gli esperimenti descritti nel Capitolo 4.

Capitolo 2

Remote Procedure Call

In questo capitolo si introducono le Remote Procedure Calls (RPC) fornendo la definizione, discutendo le problematiche introdotte da tale paradigma ed infine discutendo la differenza fra RPC di Sun Microsystem ed i più importanti ambienti di programmazione tipo RPC in particolare con Java RMI di Sun Microsystem e di CORBA.

La Remote Procedure Call o *RPC* è un'estensione, in ambiente distribuito, del meccanismo di chiamata di procedura tradizionale previsto dai linguaggi di programmazione di tipo imperativo. Il concetto di RPC nasce alla fine degli anni '70. La prima implementazione popolare di *RPC* (su UNIX) è stata la *RPC* di Sun Microsystem, la quale risale ai primi anni '80. Il modello *RPC* estende la programmazione sequenziale al contesto distribuito utilizzando il paradigma client-server. Un programma sequenziale è articolato in un insieme di procedure formate da un'unico flusso di controllo. Nel modello *RPC* abbiamo sempre un unico flusso di controllo logico, ma tale flusso è, di fatto, eseguito in un ambiente distribuito; la procedura è eseguita nel contesto di un processo su un altro elaboratore connesso in rete al calcolatore sul quale avviene la chiamata. Lo scambio di parametri, tra il programma chiamante e procedura remota chiamata, avviene con una tradizionale invocazione di un sottoprogramma. Il programmatore invoca la procedura remota con le stesse modalità con cui invoca la procedura locale,

senza dover necessariamente tener conto dei dettagli delle diverse piattaforme su cui gira la parte dell'applicazione chiamante e la parte dell'applicazione chiamata. Il modello *RPC* [6, 13, 14] si basa sull'utilizzo di procedure di interfaccia (dette stub) che implementano, in maniera trasparente al programmatore, la comunicazione su rete per lo scambio dei parametri tra il programma chiamante (client) e procedura chiamata (server); a tal fine, le procedure stub, accedono direttamente al livello di trasporto dei protocolli di rete e vengono compilate e collegate alle rispettive parti dell'applicazione.

In particolare:

1. **stub client:**
   preleva i parametri di scambio dal client, li impacchetta in un formato indipendente dalla macchina, li inserisce in un apposito messaggio, li invia tramite un software di rete che utilizza il livello di trasporto dello stack TCP/IP alla macchina dove risiede il server e si mette in attesa del messaggio di risposta. Appena giunge il messaggio di risposta il client interpreta il messaggio, estrae i parametri di uscita, restituisce i parametri appena estratti al programma chiamante, riprende la sua esecuzione; dal momento che è stato ricevuto il messaggio di risposta vuol dire che l'esecuzione della procedura remota è terminata correttamente.

2. **stub del servente:**
   rimane in attesa che il software di rete consegni il messaggio contenente la richiesta di esecuzione della procedura, preleva i parametri di ingresso dal messaggio, converte i dati nella rappresentazione standard della macchina, effettua la chiamata di procedura vera e propria. Il risultato della procedura viene impacchettato come nel caso dello *stub* del *client*, in formato indipendente alla macchina e inserito in un messaggio di risposta che viene poi inviato sfruttando il livello di trasporto di rete al client;

Le operazione svolte dagli stub per l'inserimento e l'estrazione dei parame-

tri della chiamata di procedura prendono il nome di marshalling/unmarshalling [13]. In particolare il marshalling dei dati deve tener conto di alcuni aspetti molto importanti:

- **la conversione del formato dei dati:**
  le macchine client e server possono avere differenti rappresentazioni dei dati in memoria. In particolare le due piattaforme possono essere eterogenee, di conseguenza possono adottare rappresentazioni interne diverse sia per i dati di tipo numerico che per gli altri tipi di dati (ad esempio: si può avere una macchina con ordinamento dei byte *big endian* e un'altra macchina con ordinamento *little endian*)[6]

- **serializzazione dei dati:**
  i dati vengono trasformati in sequenze di byte e impacchettati secondo un formato compreso dal client e dal server (naturalmente questo formato è indipendente dalla macchina su cui stiamo lavorando). In particolare occorre serializzare dati strutturati (esempio: array, record, etc)

Nei sistemi *RPC* la comunicazione fra processi è intrinsecamente di tipo sincrono-bloccante. Il processo chiamante rimane sospeso fino al completamento della procedura remota. Questo può essere pericoloso nel caso di guasti ai nodi o alla perdita di pacchetti nella rete. In particolare questi provocano: la perdita del messaggio inviato al server o il messaggio inviato al client contenente la risposta, la caduta del nodo server dopo la ricezione del messaggio ma prima di aver restituito la risposta. Naturalmente può guastarsi anche il nodo client dopo l'invio della richiesta ma questo non è significativo. In conseguenza a questi tipi di errore (perdita di messaggi nella rete) si hanno: ripercussioni sul risultato ottenuto cioè il processo chiamante restituisce un risultato errato; Nei casi più gravi il processo chiamante rimane in attesa dei messaggi del processo chiamato. Per risolvere questi tipi di errore al livello middleware si utilizzano politiche di timeout e ritrasmissione dei messaggi.

Le informazioni necessarie al processo chiamante, o processo client, per eseguire la chiamata di procedura remota (esecuzione dell'RPC) sono l'indirizzo del nodo su cui risiede fisicamente il processo server e il nome della procedura da invocare in remoto. Una proprietà molto importante degli *RPC* è che garantiscono la trasparenza della posizione. Il meccanismo che implementa questa caratteristica è il processo `binder` [6, 14]. Il processo `binder` è responsabile di gestire l'elenco di servizi disponibili su un determinato nodo tra cui le procedure RPC: i processi server, tramite delle opportune chiamate alla libreria RPC possono registrarsi all'elenco di questi servizi, i processi client (sempre tramite opportune chiamate alla libreria RPC) possono richiedere la lista dei servizi o verificare la disponibilità di un dato servizio su un certo nodo. Un'altra proprietà importante delle applicazioni basate sul modello *RPC* è che esse sono sviluppate mediante strumenti automatici di generazione degli `stub`: un esempio è costituito dal compilatore *rpcgen* originariamente sviluppato nel sistema *RPC* di Sun Microsystem. Una volta che sono noti i parametri dell' *RPC*, il formato dei messaggi, le procedure da eseguire in remoto la struttura degli stub è esattamente definita. Quindi, utilizzando dei generatori di codice, è possibile automatizzare la produzione delle procedure di stub. La specifica viene redatta dal programmatore mediante un linguaggio ad alto livello che viene denominato *IDL*[1] [1, 15]. La specifica in un linguaggio IDL rappresenta una sorta di contratto tra le componenti client e server di un'applicazione distribuita. Tecnicamente, un linguaggio IDL si rende necessario per poter generare automaticamente il codice degli stub e per poter collegare le procedure applicative con quelle dello strato middlware. Tuttavia la disponibilità di linguaggi IDL ha assunto nel tempo una valenza più generale, come strumento di specifica di alto livello e di documentazione dal punto di vista architetturale (statico) delle componenti di un sistema software distribuito. Per questo motivo, tecnologie middleware più recenti ed in particolare le piattaforme ad oggetti distribuiti (ad esempio CORBA) hanno riscoperto l'importanza dei linguaggi di tipo IDL. IDL può

[1] IDL: dall'acronimo Interface Definition Language

servire per generare codice per esempio f77, java, mentre nel caso di RPC di Sun genera solamente o C++. La *RPC* per come è definita, per le proprietà fin qui discusse si adatta al calcolo distribuito. Il calcolo distribuito richiede: particolare cura del protocollo di comunicazione e con *RPC* risolviamo questo problema utilizzando marshalling/unmarshalling; richiede la connessione tra utenti e risorse deve essere trasparente e *RPC* garantisce la trasparenza della rete e la trasparenza della posizione; richiede proprietà di fault tolerance e con *RPC* grazie ai metodi di ritrasmissione e time out, almeno in parte, risolviamo questo problema; infine, ma non meno importante degli altri punti con RPC riusciamo ad avere una buona scalabilità, scalabilità che di fatto varia in base al tipo di *RPC* utilizzata.

2.1 *RPC* in *C*

2.1.1 La semantica

In questo paragrafo si introducono i passi fondamentali della programmazione di un client e di un server remoto sviluppati nell'ambiente di programmazione C.

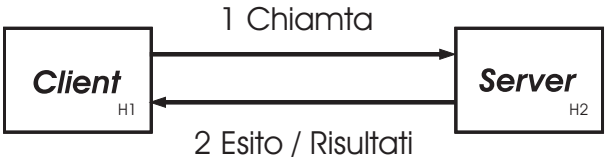

Figura 2.1: modello Client Server

La tipologia di iterazione fra client e server mostrata in figura 2.1 esegue i seguenti passi:

- il client chiama la procedura remota sul server inviandogli il nome del programma il nome della procedura, ed i parametri dopodiché il client si mette in attesa dei risultati;

- il server decodifica il codice della procedura ed esegue i seguenti passi:
  - invoca il servizio richiesto dal client;
  - esegue tale servizio;
  - calcola il risultato/i risultati;
- il server invia i risultati ottenuti al client;
- il client, una volta ricevuti i dati, si sblocca continuando la sua esecuzione;

La semantica riportata è indipendente dai dettagli relativi al passaggio di informazioni da un host all'altro: tali dettagli sono incapsulati nelle funzioni che permettono l'esecuzione di un RPC implementati dal supporto a tempo di esecuzione. Compito del programmatore è specificare:

- il nome dell'host dove è in esecuzione il server;
- **nome del programma:** il nome del programma ha la seguente sintassi `program nome-programma{...} = numero-identificatore` dove il `nome-programma` è identificato univocamente dal `numero-identificatore`
- **numero di versione del programma:**il numero di versione del programma ha la seguente sintassi `nome-versione{...}=numero-versione` dove `nome-versione` viene utilizzata nella dichiarazione del client e del server e il `numero-versione` (di solito si utilizza un numero naturale che parte da 1) è il numero della versione del programma;
- **il nome delle procedure da esportare in remoto:** queste procedure hanno la seguente sintassi `tipo-risultato nome-procedura(tipo-parametro) = numero-procedura` dove `tipo-risultato` e `tipo-parametro` sono i tipi dei parametri passati alla procedura, e il `numero-procedura` è il numero d'ordine della procedura all'interno delle procedure esportate da questo programma.

- **protocollo di trasporto utilizzato (TCP oppure UDP);**

Maggiori dettagli dei punti sopraindicati vengono dati nel prossimo paragrafo in particolare nella descrizione del `file.x`

Il discovery nel senso più generale (cioè discovery distribuito) non è supportato dall'astrazione RPC, ma deve essere implementato a livello applicazione dal programmatore. In questo paragrafo ci si riferisce all'attività di discovery di un servizio (procedura remota) su un dato host (di cui si conosce l'inidirizzo). Nel paragrafo precedente è stato descritto il processo `binder`, nel caso di *RPC* di Sun il processo `binder` e proprio il portmap. Il portmap è un servizio di rete che parte all'accensione della macchina ed è in esecuzione su una porta dedicata (normalmente la porta 111). Altri servizi di rete si possono registrare al portmap. In particolare un programma server si registra presso il portmap dell'host su cui è in esecuzione fornendogli il nome del programma, la versione e il numero di porta sul quale accetta richieste di esecuzione di procedure vedi figura 2.2. Naturalmente in tutti questi passi, come viene descritto nella pagina precedente, il client comunque deve sapere il nome dell'host del quale deve interagire il portmap.

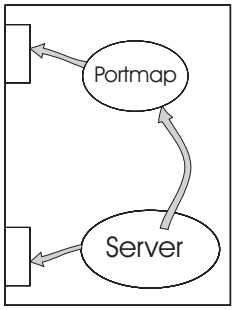

Figura 2.2: il server si registra sul portmapper.

Come possiamo notare nella figura 2.3 il programma client interroga il portmap presente sull'host del server. Se il programma richiesto è registrato allora il portmap risponderà al client inviandogli il numero di porta su cui è in attesa il server. Nel passo successivo il client comunica direttamente con il server figura 2.4.

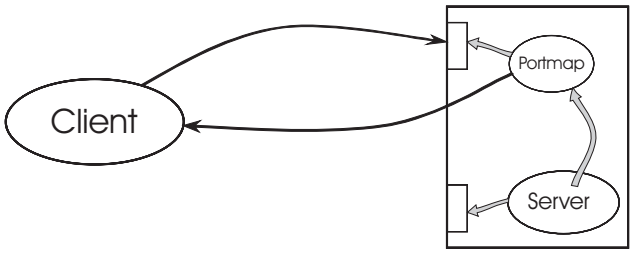

Figura 2.3: il client interroga il pormapper.

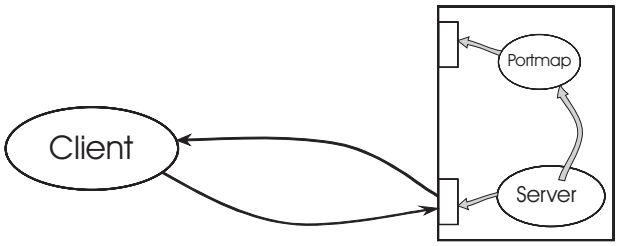

Figura 2.4: comunicazione tra client e server.

2.1.2 Implementazione del *RPC*

In questo paragrafo si introducono tutti per eseguire un'applicazione RPC di Sun gli strumenti necessari e proposti alcuni semplici esempi. Il primo strumento necessario è l'IDL definito nei precedenti paragrafi. Nel caso specifico di RPC di Sun l'IDL è il linguaggio con cui scriviamo il `file.x`. Infatti la sua compilazione genera tutti i file necessari ad avere un'interfaccia tra client e server. Di seguito vengono descritti il compilatore `rpcgen` ed i file generati da *file.x*.

Compilazione con `rpcgen`

`rpcgen` [6, 13] è un compilatore il quale accetta un servizio, cioè un file con estensione `.x` (esempio `file.x`), definita in un linguaggio IDL molto simile al C e come risultato genera i seguenti file scritti in linguaggio C:

1. lo skeleton del server (`file_svc.c`);
2. o skeleton del client (`file_clnt.c`);

3. una routine XDR (`file_xdr.c`);

4. un header file (`file.h`);

I file generati dalla compilazione del `file.x` da soli non bastano, infatti l'utente deve scrivere anche i seguenti file: `client.c` e `server.c`. L'unica attenzione è che in questi file deve essere incluso il file `file.h`. Successivamente l'utente potrà compilare (con il compilatore `gcc`) inserendo i rispettivi file di skeleton in questo modo: il file `client.c` con il `file_clnt.c` e il file `server.c` con il `file_svc.c`. Viene ora mostrato un esempio di compilazione prima con il compilatore *rpcgen* poi con quello tradizionale del linguaggio di programmazione C (`gcc`):

```
cecchi-> rpcgen file.x
cecchi-> gcc  file_clnt.c mainClient.c -o client
cecchi-> gcc  file_svc.c server.c -o server
```

Una volta compilato non rimane che eseguire il client ed il server su due host distinti. Nei prossimi paragrafi vengono descritte in dettaglio le funzionalita' di tutti i file compresi quelli generati dalla compilazione con `rpcgen`.

Descrizione del `file.x`

`file.x` è un file scritto in un linguaggio IDL molto simile alla sintassi del C, e permette la generazione dei file che servono ad interfacciare il client ed il server. Contiene il nome simbolico del programma, la descrizione delle procedure, la descrizione delle strutture dati da pubblicare come RPC. Vediamo un semplice esempio:

```
const COSTANTE = 1024;
program NOME_PROGRAMMA{
  version NUMERO_VERSIONE{
    int esempioDiProcedura(int) = 1;
```

```
  } = 1;
} = 0x20000007;
```

In particolare:

1. Il file inizia con la dichiarazione (facoltativo) di costanti ed eventuali strutture dati utilizzate per inviare i parametri tra client e server attraverso la rete. L'esempio mostra la dichiarazione di una costante `const COSTANTE = 1024`;

2. contiene `program NOME_PROGRAMMA = 0x20000007` dove il `NOME_PROGRAMMA` è il nome che verrà utilizzato per istanziare il codice del client e del server, l'identificatore `0x20000007` è l'identificatore unico del programma usato nella rete. L'utente può utilizzare identificatori che hanno valori compresi da `0x20000000` a `0x3fffffff`. Gli identificatori che vanno da `0x0` a `0x1fffffff` sono utilizzati da Sun, invece quelli maggiori di 0x3fffffff sono riservati per altri servizi non utente. L'identificatore di programma usato nella rete non identifica univocamente il programma all'interno del portmap, per questo e necessario anche il numero di versione;

3. il numero di versione ha il seguente formato:*version NUMERO_VERSIONE*{ *...* } = *1*questo permette la coesistenza di più versioni dello stesso programma. Esempio aggiungendo una procedura in un determinato host, basta inserirla nel al programma contenuto nel `file.x` e aggiornare il numero di versione di tale programma, non occorre fare alcuna modifica sul resto degli host.

4. Infine vengono riportati i prototipi delle procedure implementate nella forma: `int esempioDiProcedura(int) = 1` che descrive in dettaglio quali tipi di dato coinvolti, il nome stesso della procedura;

Descrizione del `server.c`

Il file `server.c` contiene il codice eseguito sul server. Questo file è scritto interamente da parte dell'utente. La sua struttura, come il resto dei file generati dal compilatore `rpcgen`, deve seguire alcune regole di impostazione che deve assolutamente rispettare per non incorrere in errori nella fase di compilazione. Quindi deve: includere il `file.h`; l'intestazione delle procedure scritte all'interno del file `server.c` devono avere la forma `nomeProcedura_numeroVersione_svc()` dove: `nomeProcedura` è lo stesso utilizzato sia dal client che quello dichiarato nel `file.x`; `numeroVersione` deve essere lo stesso di quello utilizzato nel client altrimenti sono viste come due procedure diverse. Di seguito viene fatto un esempio di `server.c` per chiarire come è la struttura del file:

```
 #include "file.h"
int esempioDiProcedura_1_svc(int *argp,struct svc_req *rqstp)
 {
   int retval;
   /* qui va inserito il codice del server */
   return retval;
 }
```

Descrizione del `file_svc.c`

Nel `file_svc.c` sono presenti tutte le procedure necessarie al server per registrarsi sul portmap ed aprire un socket dove riceverà le richieste da parte del client. Questo file è generato dalla compilazione del `file.x` con `rpcgen`. Nella successiva fase di compilazione con il compilatore gcc questo file viene compilato insieme al file `server.c`. Al suo interno contiene il `main` che verrà eseguito dal lato server il quale contiene:

- la procedura `pmap_unset(NOME_PROGRAMMA,NUMERO_VERSIONE)` che è un'interfaccia verso il servizio portmap la quale distrugge tutti i map-

ping fra le triple `[ NOME_PROGRAMMA,NUMERO_VERSIONE, *]` e le porte sui servizi portmap di quella macchina;

- le procedure `transf = svcudp_create(RPC_ANYSOCKET)` e `transf = svctcp_create(RPC_ANYSOCKET,0,0)` che creano un socket TCP e uno UDP dove il server riceverà le richieste da parte del client al loro interno contengono la procedura `bindresvport` che cerca una porta libera dove aprire il socket, la `bind` e la `listen` per accettare le connessioni;

- la procedura `svc_register(transp, NOME_PROGRAMMA, NUMERO_VERSIONE, file_prog_1,...)` che effettua la registrazione di questo servizio sul portmap in tutti e due i casi (UDP TCP).

- la procedura `svc_run()` la quale si mette in ascolto delle richieste che fara' il client.

Di seguito viene proposto un esempio:

```
 /* Please do not edit this file.
 * It was generated using rpcgen.*/
#include "file.h"
#include <rpc/pmap_clnt.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
...
#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif
static void
file_prog_1(int /*...-*/, register SVCXPRT *transp)
{
  union {/* ..... */} argument;
  union {/* ..... */} result;  /* ..... */
```

```
  switch (rqstp->rq_proc) {
    case NULLPROC: /* ..... */ return;
    case esempioDiProcedura:/* ..... */ break;
    default:
    svcerr_noproc (transp);
    return;
  } /* ..... */
}
int main (int argc, char *argv[]){
  register SVCXPRT *transp;
  pmap_unset (NOME_PROGRAMMA, NUMERO_VERSIONE);
  transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL) {
    fprintf (stderr, "%s", "cannot create udp service.");
    exit(1);
  }
  if (!svc_register(transp, NOME_PROGRAMMA, NUMERO_VERSIONE,
  file_prog_1, IPPROTO_UDP)) {
    fprintf (stderr,"%s","unable to register(NOME_PROGRAMMA,
                                    NUMERO_VERSIONE,udp).");
    exit(1);
  }
  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {
    fprintf (stderr, "%s", "cannot create tcp service.");
    exit(1);
  }
  if (!svc_register(transp, NOME_PROGRAMMA, NUMERO_VERSIONE,
  file_prog_1, IPPROTO_TCP)) {
    fprintf (stderr,"%s","unable to register(NOME_PROGRAMMA,
                                     NUMERO_VERSIONE,tcp).");
```

```
    exit(1);
  }
  svc_run ();
  fprintf (stderr, "%s", "svc_run returned");
  exit (1); /* NOTREACHED */
}
```

Descrizione del `file_clnt.c`

Il `file_clnt.c` viene utilizzato per creare un'interfaccia con il server. Contiene le chiamate di procedura che verranno eseguite in remoto. Queste procedure hanno la forma `nomeprocedura_numeroVersione` (notare l'analogia con le procedure dichiarate nel `server.c` che sono della forma `nomeprocedura_numeroVersione_svc()`). La `nomeprocedura_numeroVersione` al suo interno esegue la procedura `clnt _call(clnt, procnum, inproc, in, outproc, out,tout)` la quale chiama la procedura `procnum` associata al handle del client `clnt` ottenuto dalla creazione di un client RPC tramite la `clnt_create` Inoltre implementa una politica basata su timeout per la gestione degli errori.

Vediamo un esempio:

```
  /*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include <memory.h> /* for memset */
#include "array.h"
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };
enum clnt_stat
esempioDiProcedura_1(int *argp, CLIENT *clnt){
  return (clnt_call(clnt, esempioDiProcedura, .... ,TIMEOUT));
}
```

Descrizione del `client.c`

Il file `client.c` è scritto dall'utente. Contiene la chiamata di procedura `cliente clnt_create(host,NOME_PROG,NUMERO_VERSION,protocol);` dove:

1. `host:` è il nome dell'host sul quale eseguire in remoto le procedure dichiarate in `NOME_PROG`;
2. `NOME_PROG:` è il nome del programma descritto nel `file.x`;
3. `NUMERO_VERSION:` è il numero di versione del programma descritto nel `file.x`;
4. `PROTOCOL:` è il tipo di protocollo di comunicazione scelto (*TCP* oppure *UDP*);

```
#include"file.h"
int main(int argc, char *argv[])
{
  CLIENT *client;
  int calcolare,risultato;
client=clnt_create(host,NOME_PROGRAMMA,NUMERO_VERSIONE,"tcp");
  risultato = esempioDiProcedura_1(calcolare,client);
  return risultato;
}
```

La procedura `clnt_create` permette di connettersi al portmap dell'host passato come parametro, interroga il portmap per vedere se il programma `NOME_PROGRAMMA` con la versione `NUMERO_VERSIONE` è presente. Se è presente restituisce la struttura `CLIENT` che contiene il numero di porta sul quale è in ascolto il server altrimenti fallisce. Tramite la chiamata della procedura vera e propria (esempio `calcola_1` notare che 1 è la versione) è eseguita la procedura in remoto connettendoci direttamente sul server remoto per mezzo della `clnt_call`.

Descrizione del `file_xdr.c`

Il `file_xdr.c` viene generato dalla compilazione del `file.x`. Contiene le procedure che traducono le strutture dati definite nel `file.x` in un formato indipendente alla macchina. Nel caso dell'esempio presentato eseguono fanno niente di interessante dato che non ho dati strutturati. Se prendiamo l'esempio del file `array.x` presente nell'appendiceA, esso contiene due strutture `struct ingresso, struct uscita` entrambe formate da due campi: un array di `double` e un `intero`. Se analizziamo il file `array_svc.c` al suo interno contiene la procedura `xdr_vector(xdrs,(char` × `)obj->vectorIN,DIMENSIONE, sizeof(double),(xdrproc_t)xdr.double)` che traduce il vettore qdi double `vectorIN` in un formato xdr. Infatti ha come parametro `(char*)obj->vectorIN` che è l'indirizzo del puntatore dell'array, `DIMENSIONE` è la dimensione dell'array, `sizeof(double)` è' la dimensione dell'elemento dell'array, `(xdrproc_t)xdr.double` è la traduzione degli elementi dall'array in C al formato xdr.

Questo paragrafo ci porta a concludere che programmare con RPC è sufficientemente semplice. Il lavoro più impegnativo è la scrivere il file con l'IDL. La sua compilazione genera, in modo automatico, tutto quello di cui il programmatore ha bisogno per eseguire un programma in ambito distribuito. Questo è reso semplice dal fatto che RPC esegue le chiamate delle procedure allo stesso modo delle procedure locali.

2.2 RPC in Java

In questo capitolo si introducono le Java RMI. Inoltre vengono discusse le principali differenze tra RPC e JavaRMI di Sun Microsystem.

Java RMI [17, 16] utilizza l'ambiente di elaborazione omogeneo costituito dalla Java Virtual Machine (JVM). In Java RMI un'applicazione è costituita da processi server che creano oggetti accessibili remotamente, registrandone il nome nell'apposito registro *RMIregistry* e si pongono in attesa di invocazione dei metodi sugli oggetti remoti da parte dei clienti. Un client ottiene

il riferimento ad un oggetto, effettuando una ricerca nel registro *RMIregistry* sull'host remoto e successivamete invoca i metodi remoti. È il middleware RMI che si occupa dei dettagli della comunicazione: agli occhi del utente l'invocazione remota appare come un'invocazione standard di un metodo su un oggetto nel linguaggio Java tradizionale. Nell'invocare un metodo, è possibile scambiare oggetti come parametri o valori di ritorno; RMI fornisce anche i meccanismi necessari per caricare su richiesta il codice di un oggetto, e per serializzarne e trasmetterne lo stato. Lo strato middleware della piattaforma Java basato sulla tecnologia RMI (Remote Method Invocation) è molto simile al middleware RPC.

2.2.1 Differenze tra RPC ed RMI

In questo paragrafo vengono definite le differenze sostanziali tra RPC ed RMI. Inoltre, come è stato fatto nel paragrafo precedente, viene descritto un esempio di java RMI.

Compito dell'utente

L'utente ha il compito di scrivere una classe client, nell'esempio proposto questa classe è contenuta nel file `RMIClient.java`, e contiene le chiamate ai metodi remoti. Inoltre l'utente deve scrivere una classe server anche in questo caso inserita nel file `RMIServerImpl.java` nella quale troviamo oggetti che saranno eseguiti in remoto come in RPC (in quel caso i file erano `client.c` e `server.c`). La struttura dei due programmi è conforme allo stile di programmazione java.

Vediamo un esempio:

```
import java.rmi.*;
...
public class RMIClient {
  public static void main(String[] args) {
    try {
```

```
        /*dichiarazione strutture dati*/
        System.out.println("Cerco il server...");
        RMIServer server = (RMIServer)
          Naming.lookup("rmi://"+args[0]+"/home/EsempiJava");
        System.out.println(" ecco il Server ...");
        ...  c = server.calcola(a,b); ...
        }
        catch (Exception e) {
          e.printStackTrace();
        }
    }
```

Rispetto ad un programma scritto in java tradizionale è necessario importare i nomi del package `java.rmi`. La classe `RMIClient` contiene il metodo `RMIServer server = (RMIServer) Naming.lookup(rmi:  nomeHost)` il quale interroga l' `rmiregistry` per vedere se il servizio `RMIServer` è in esecuzione sull'host `nomeHost` come in RPC il `client` attraverso la `clnt_create(...)` interroga il portmap sull'host passato come parametro; attraverso l'invocazione del metodo `...  = server.calcola(...)  ...` (notare che per l'utente è come chiamare un metodo tradizionale java in locale) avviene la connessione alla diretta alla classe RMIServer sull'host remoto come in RPC si utilizza la chimamta `...esempioDiProcedura(...)....`

Vediamo ora l' esempio del server:

```
import java.rmi.*;
import java.rmi.server.*;
...
spublic class RMIServerImpl extends UnicastRemoteObject
                                    implements RMIServer {
  public static void main(String[] args) {
    try {
      RMIServerImpl server = new RMIServerImpl();
```

```
    Naming.rebind("home/EsempiJava", server);
     System.out.println("Inizio del Server");
  } catch (Exception e) { e.printStackTrace(); }
}
public RMIServerImpl() throws RemoteException {}
public void print(String s) throws RemoteException {
  System.out.println(s);
}
public int calcola(int a, int b) throws RemoteException{
  try {   ...   }
}catch (Exception e) { e.printStackTrace(); return ...; }
```

In questo caso le differenze sostanziali consistono nell'importare i nomi dei package `java.rmi` e `java.rmi.server`, e nel chiamare il metodo `Naming.rebind(rmi://host:port/oggetto, server)` che comunica all'rmiregistry che tale oggetto è disponibile. Come possiamo notare nell'esempio sopra indicato la classe `RMIServerImpl` deve estendere la classe `UnicastRemoteObject`. Inoltre non meno importante tutti i dati coinvolti devono essere serializzati. Quando si instaura un collegamento virtuale tra stub e skeleton (o stub e server) questo è un tipo di collegamento sequenziale, per questo tali dati devono essere serializzati.

Compilazione con rmic

Per supportare la trasparenza della rete in JavaRMI come in RPC c' è un compilatore, `rmic`, che genera lo `stub` e lo `skeleton` dell'oggetto remoto . L'unica differenza tra i due compilatori è che `rpcgen` compila un file scritto in un linguaggio simile al C (`file.x`) mentre `rmic` compila direttamente i file (server) generati dal compilatore `javac` (`file.class`). Notare che in questo caso l'IDL è proprio il linguaggio java. Una volta avvenuta la compilazione per eseguire il client e il server si utilizza il comando `java nome_file_da_eseguire` come nelle tradizionali applicazioni java. Dalla ver-

sione 2.0 del JDK il file di skeleton non viene più generato. Lo stub generato da `rmic` fornisce una simulazione locale sulla JVM[2] del client dell'oggetto remoto, inoltre non viene inviato direttamente l'oggetto ma una rappresentazione serializzata e successivamente, sulla JVM target viene ricreata una copia identica dell'oggetto (deserializzazione).

Rmiregistry

Nel caso di Java RMI come in RPC viene utilizzatto un servizio `binder`, per avere a disposizione tutti gli oggetti disponibili su un nodo, `rmiregistry`. `rmiregistry` è un'applicazione che si occupa di registrare all'interno di un registro il nome logico dell'oggetto remoto dalla *classe Naming* presente nel package `java.rmi`. Come nel caso del `portmap` *RPC* l'oggetto server si registra su `rmiregistry` e il client può interrogare tale applicazione per vedere se l'oggetto è disponibile su quel nodo. Come il `portmap` di *RPC*, `rmiregistry` e sempre in ascolto su una porta, che di default è la 1099.

Le differenze sostanziali tra *java RMI* e *RPC* descritte in questo paragrafo sono: *java RMI* è un linguaggio ad oggetti mentre *RPC* no; in entrambi i casi la comunicazione non è prerogativa del programmatore; entrambi sfruttano sistemi di stub e skeleton per interfacciare l'applicazione client e l'applicazione server; entrambe utilizzano un servizio di registrazione dove vengono registrate i metodi/procedure del server.

2.3 RPC con CORBA

In questo paragrafo si introduce la definizione di CORBA. CORBA [3] [1, 8, 15] è la specifica di un modello di riferimento per middleware ad oggetti. CORBA fa parte di un più ampio modello architetturale denominato OMA [4]. CORBA e OMA sono definiti dal consorzio OMG [5] [10]. OMG nasce nel 1989 come

[2] Java Virtual Machine

[3] acronimo di Common Object Request Broker Architecture

[4] OMA: dall'acronimo Object Management Architecture

[5] OMG: acronimo di Object Management Group

un'organizzazione senza profitti per iniziativa di otto aziende importanti a livello mondiale tra queste le più importanti Canon, HP (Hewlett-Packard), Philips Telecominications, Sun Microsystem e Unisys. OMG si pone come obiettivi principali la promozione e standardizzazione di tecnologie e metodologie relative a sistemi software distribuiti orientati agli oggetti, in modo da ridurne la complessità di sviluppo e supportare l'interoperabilità tra ambienti eterogenei. La caratteristica più importante dell'OMG è che non produce software ma specifiche che servono a guidare i produttori nella costruzione di piattaforme middleware che supportino l'interoperabilità, risolubilità e portabilità dei componenti software su sistemi eterogenei. CORBA è la specifica di un middleware non proprietario che, oltre alle caratteristiche delle piattaforme di tipo DOM [6] [2], offre diversi vantaggi per lo sviluppo e l'integrazione di applicazioni distribuite. Le proprieta' piu' importanti dello standard CORBA sono:

- supporta le tecniche di ingegneria del software a oggetti e a componenti;
- supporta molti linguaggi di programmazione;
- consente lo sviluppo e l'integrazione di applicazioni multilinguaggio;
- definisce interfacce standard per un ricco insieme di servizi di base e avanzati;
- prevede l'interoperabilità tra implementazioni di produttori differenti;
- infine è riconosciuto da oltre 700 aziende che fanno parte di OMG.

Come mostrato all'inizio del paragrafo CORBA fa parte del modello Architetturale OMA. Nella figura 2.5 viene analizzata l'architettura OMA dove possiamo notare che è la specifica a piu' alto livello dell'OMG.

Prevede quattro ambiti di standardizzazione per servizi di supporto alle applicazioni: l'ORB[7] i servizi CORBA Service i servizi ad alto livello CORBAfacilities i domini CORBAdomains. Il termine CORBA si riferisce in

[6]DOM: acronimo di Document Objrct Model

[7]ORB: dall'acronimo Object Request Broker

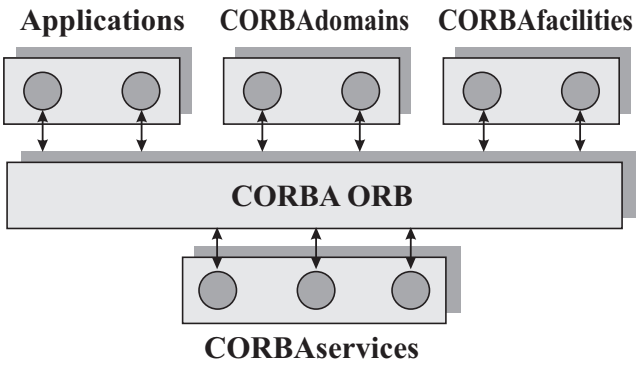

Figura 2.5: Modello di riferimento OMA.

senso stretto alla specifica dell'ORB; in senso ampio è frequentemente usato per indicare tutta l'architettura OMA. In figura 2.6 sono rappresentati gerarchicamente a piramide i quattro elementi di OMA. La figura 2.6 è stata inserita per evidenziare il livello crescente di astrazione dei servizi offerti. Analizzando la figura 2.6 si vede che alla base della piramide c' è l'ORB,

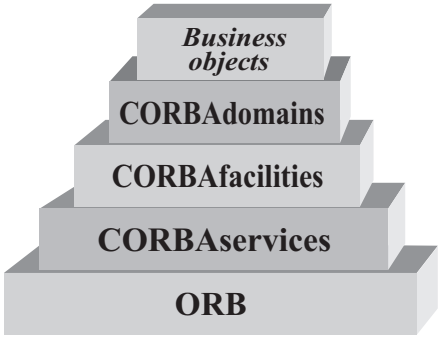

Figura 2.6: Livelli di astrazione dei componenti OMA.

l'ORB ha lo scopo di fornire i meccanismi di comunicazione fondamentali. Il livello superiore della piramide contiene i CORBAservice contiene i servizi agli oggetti, mentre il livello ancora superiore contiene i CORBAfacilities dove al suo interno troviamo i servizi alle applicazioni di tipo orizzontale utili alle diverse categorie di applicazioni. Il livello dei CORBAdomains (anche detti facilities verticali) contiene i servizi specializzati per domini applicativi, quali quello delle telecomunicazioni, quello finanziario e assicurativo, quello

sanitario. In cima alla piramide sono i *business objects*, termine con cui si indicano componenti riusabili di livello applicativo. Il nucleo dell'architettura è l'ORB, che fornisce un insieme di servizi per l'invocazione di metodi su oggetti remoti figura 2.7. L'ORB è l'intermediario di ogni iterazione, e garantisce le proprietà di trasparenza dei sistemi operativi, dei protocolli di rete dei linguaggi di programmazione, e della posizione e migrazione degli oggetti. Trattandosi di uno standard, l'ORB favorisce la trasparenza delle implementazioni comerciali, nel senso che le applicazioni sono portabili sugli ORB conformi allo standard. Un apposito standard per l'interoperabilità tra ORB garantisce inoltre la possibilità di costruire applicazioni, in cui gli oggetti sono distribuiti su più implementzioni differenti dell'ORB. La rappresentazione tipica dell'ORB mostrata in figura 2.7 non deve fuorviare: è bene osservare esplicitamente che si tratta di una rappresentazione *logica* e puramente *funzionale*.

Figura 2.7: Rappresentazione logica dell'ORB.

L'ORB non è un'entità fisicamente separata dal client e dall'oggetto, ma consiste di sottoprogrammi che vanno compilati e collegati (staticamente o dinamicamente) al cliente e all'oggetto, a tempo di esecuzione fanno parte delle immagini dei rispettivi processi. Al programmatore l'ORB si presenta come un insieme di interfacce applicative per l'invocazione dei metodi remoti. Fondamentale per l'indipendenza dai linguaggi di programmazione è la presenza nello standard CORBA di un linguggio di definizione delle interfacce, noto come OMG IDL. Tutti gli oggetti che forniscono servizi tramite l'ORB hanno un'interfaccia specificata in IDL, siano essi oggetti applicativi o appartenenti all'infrastruttura CORBA. La trasparenza del linguaggio è ottenuta mediante regole standard di traduzione da IDL ai piu' diffusi linguaggi di

programmazione. Servizi, facilities e domini OMA sono illustrati in maggior dettaglio nella figura 2.8.

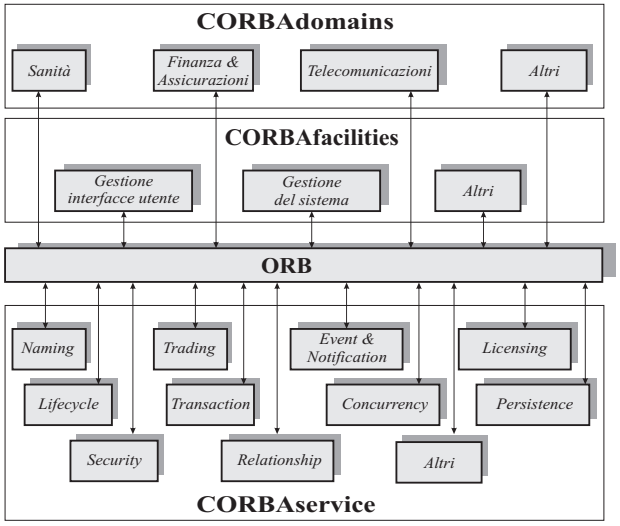

Figura 2.8: Servizi facilities e domini OMA.

Ciascun componente raffigurato in un rettangolo consiste di un insieme di oggetti dotati di interfacce IDL standard, cui le applicazioni accedono tramite l'ORB. I CORBAservices formano un insieme di servizi per gli oggetti: localizzazione degli oggetti (Naming e Trading); gestione eventi (Event e Notification); gestione licenze (Licensing); creazione, cancellzione e migrazione di oggetti (Lifecycle); gestione di transazioni (Transaction); controllo della concorrenza (Concurrency control); sicurezza (Sicurity); gestione delle associazioni tra oggetti (Relationship); I sevizi possono essere classificati in servizi di base e servizi avanzati. Tra i servizi di base rientrano quelli di Naming e Trading per la trasparenza della posizione: il primo offre meccanismi per localizzare gli oggetti sull'ORB in base al nome, similmente alle pagine bianche telefoniche; il secondo si basa su un approccio analogo a quello delle pagine gialle telefoniche, consentendo di localizzare oggetti in base alle loro caratteristiche. Tra i servizi di base annoveriamo anche quelli di gestione eventi. Gli altri servizi possono essere considerati più avanzati; i principali tra essi (Licensing, Lifecycle, Transaction, Concurrency, Persistence e Secu-

tity). CORBAfacilities e CORBAdomains sono gli elementi dell'architettura OMA che offrono le maggiori potenzialità in termini di supporto per l'integrazione e cooperazione applicativa. Il loro processo di standardizzazione può essere considerato ancora agli inizi, e vede coinvolte anche industrie, enti e altri consorzi di riferimento nei relativi settori, di cui l'OMG prevede di incorporare i relativi standard. È opportuno in questa sede soffermarsi sul modello a oggetti CORBA, in quanto i principi dell'orientameto agli oggetti sono visti in CORBA nella prospettiva di programmazione in ambiente distribuito, e assumono quindi accezioni differenti dal caso della programmazione in ambiente centralizzato. I concetti essenziali dell'orientamento agli oggetti in CORBA sono:

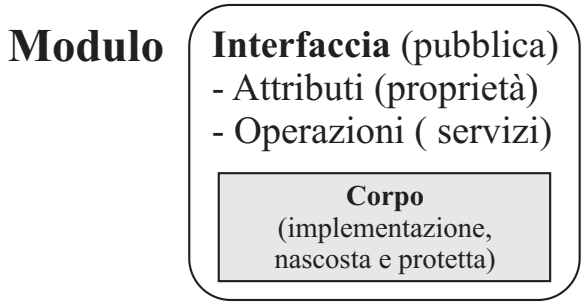

Figura 2.9: incapsulamento.

- L'incapsulamento: In figura 2.9 consiste nell'accorpare le proprietà e le responsabilità di un modulo software, e in una chiara separazione tra l'interfaccia del modulo, visibile all'esterno, e la sua implementazione, nascosta all'interno. Le proprietà rappresentano attributi o caratteristiche del modulo, e le responsabilità sono i servizi che il modulo mette a disposizione. L'interfaccia racchiude proprietà e responsabilità, e definisce le modalità con cui esse sono presentate agli altri moduli. Nel modello a oggetti di CORBA, un modulo software è un oggetto, e la sua interfaccia (descritta in linguaggio IDL) costituisce un contratto tra l'oggetto stesso e gli altri oggetti costituenti l'applicazione distribuita, che nei suoi confronti si comportano da utilizzatori dei servizi offerti.

Nell'accezione CORBA un oggetto è un'entità incapsulata che fornisce servizi ai clienti

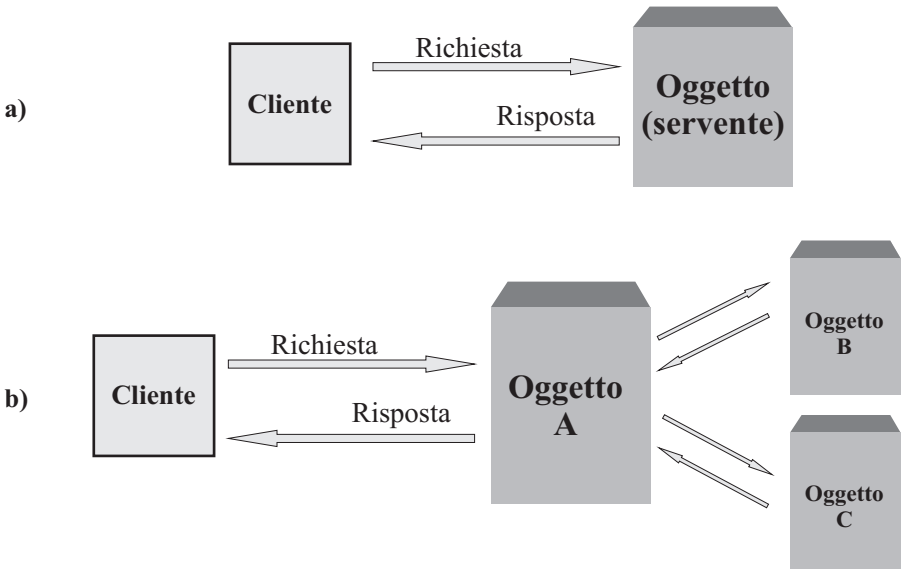

Figura 2.10: Oggetti CORBA

vedi figura 2.10 a; per espletare un servizio, un oggetto può richiedere i servizi di altri oggetti, comportandosi a sua volta come cliente nei loro riguardi vedi figura 2.10 b. Questa definizione è sostanzialmente differente da quella di oggetto nei linguaggi di programmazione. Nei linguaggi a oggetti non fortemente tipizzati come smaltalk, un oggetto è l'accorpamento di una struttura dati e delle operazioni possibili su di essa. Nei linguaggi tipizzati come C++ e Java, un oggetto è un'istanza di una classe; una classe è a sua volta un costrutto linguistico per la definizione di tipi di dati astratti. In entrambi i casi un oggetto ha natura temporanea e prettamente locale all'applicazione in cui viene creato. In CORBA un oggetto è un modulo software che possiede la proprietà e realizza i servizi definiti in un'interfaccia; è potenzialmente persistente e la sua posizione è trasparente agli utilizzatori. Una conseguenza di tale accezione è che, contrariamente a come sovente si tende a pensare, le proprietà dichiarate in un interfaccia non sono da riguardare come

strutture dati incapsulate nell'oggetto che lo implementa, un attributo non va pensato come una variabile: il suo valore potrà effettivamente essere memorizzato in una o più variabili dell'oggetto servente, o anche essere calcolato da altri dati interni all'oggetto e non specificati nell'interfaccia. Ciò che interessa è che l'oggetto che implementa una data interfaccia esibisca le proprietà e le operazioni in essa definite.

- L'ereditarieta' è un principio base delle tecniche orientate agli oggetti, che consente definire nuove entità per specializzazione o estensione di altra, e in modo incrementale. Nei linguaggi tipizzati come C++ e Java, l'ereditarietà è il meccanismo con cui si crea una nuova classe a partire da una classe base esistente (o da più classi base, nel caso di ereditarietà multipla, suportata da C++ e non da Java) la nuova classe eredita attributi e operazioni della classe base, eventualmente riferendoli o specializzandoli. Definiamo ereditarietà d'implementazione (implementation heritance) il concetto di ereditarietà appena descritto, che differisce sostanzialmente da quello che si ha in CORBA.

- L'ereditarietà in CORBA è infatti un'ereditarietà tra interfaccie (interface inheritance). La differenza non deve sorprendere: CORBA infatti non è un linguaggio di programmazione, ma un middleware di connettività tra moduli di sistemi software distribuiti; in quanto tale, non tratta meccanismi linguistici per definire tipi di oggetti, come le classi C++ e Java, ma la definizione delle interfacce tra moduli. Il linguaggio per la descrizione delle interfacce OMG IDL prevede un apposito meccanismo di ereditarietà per derivare un'interfaccia da una o piu' interfacce preesistenti.

- Il polimorfismo è un altro dei principi cardine dell'orientamento agli oggetti; esso supporta la proprietà di estendibilità di un sistema software. Per polimorfismo si intende la proprietà di un'entità di assumere varie forme, e può essere definito la capacità di oggetti diversi di assumere comportamenti differenti in corrispondenza di una stessa richiesta (ope-

razione). In altre parole, il polimorfismo è il meccanismo che consente a oggetti diversi di rispondere in modo differente a uno stesso messaggio. In CORBA il polimorfismo è intrinsecamente connesso alla definizione di oggetto, e non è legato alla creazione di gerarchie di tipi di oggetti (ovvero interfacce IDL, nella fattispecie). Un oggetto è infatti un'entità che esibisce le proprietà e i servizi definiti in un interfaccia; questa tuttavia ne descrive solo le caratteristiche e non l'implementazione. Ogni oggetto è perciò libero di implementare alla propria maniera i servizi dell'interfaccia. In effetti, un cliente richiede un servizio a un oggetto solo sulla base di informazione racchiuse nell'interfaccia, senza conoscere gli oggetti che la implementano. Una conseguenza di ciò è che CORBA non ha senso ridefinire in un'interfaccia derivata le operazioni delle interfacce.

- L'instanziazione è il processo di creazione degli oggetti di un applicazione distribuita CORBA. A ciascun oggetto l'ORB assegna un riferimento che individua univocamente l'oggetto stesso sulla rete, denominato OR (object Reference). In sostanza, il modello a oggetti di CORBA consiste di:

  - Oggetti: un oggetto è un'entità incapsulata che fornisce servizi ai clienti, identificata da un riferimento;
  - Interfacce: un'interfaccia è una specifica dei servizi che un oggetto fornisce e che un cliente può richiedere;
  - Operazioni: un'operazione è un servizio specificato in un'interfaccia e offerto da un oggetto;
  - Richieste: una richiesta è un'azione intrapresa da un cliente e rivolta ad un oggetto servernte, contenente informazioni sull'operazione da eseguire e sui dati o sugli oggetti su cui deve essere svolta; tecnicamente, una richiesta è un oggetto di sistema che viene creato istanziando un tipo di sistema (CORBA::Request).

– Tipi: i dati scambiati all'atto della richiesta e del ritorno dall'esecuzione di un'operazione devono essere tipizzati, devono cioè appartenere a un tipo, noto staticamente o eccezionalmente a tempo di esecuzione. I tipi dei dati possibili sono definiti nel linguaggio IDL, che prevede un insieme di tipi base direttamente utilizzabili, nonché la possibilità di costruire tipi d'utente. I tipi definiti in IDL, sono tradotti nel linguaggio di implementazione degli oggetti base alle regole definite dallo standard.

INTEROPERABILITÀ TRA ORB

Nel modello CORBA un'applicazione su larga scala è costituita da clienti i oggetti che, in generale, risiedono su ORB di produttori differenti. Infatti, non solo nel caso di applicazioni che coinvolgono piu' organizzazioni, ma anche all'interno di una stessa organizzazione (per esempio, un'impresa multinazionale), è inverosimile ipotizzare che nelle varie divisioni (aziende, stabilimenti, sedi, reparti o altro) siano installati ORB omogenei. È possibile che un cliente su un ORB richieda su un ORB differente; si pone dunque il problema di garantire l'interoperabilità tra implementazioni commerciali eterogenee dello standard CORBA. Lo standard definisce un modello architetturale per l'interoperabilità tra oggetti su ORB diversi, noto come CORBA Interoperability Architetture. Osserviamo esplicitamente una tale architettura non prevede interfacce applicative per la programmazione dal lato client ne dal lato server: clienti e oggetti interagiscono con l'ORB, non con la rete, e l'interoperabilità tra ORB è del tutto trasparente al programmatore.

Capitolo 3

Implementazione del prototipo MasterSlave

In questo capitolo viene definito il modello Master/Slave, viene illustrata l'architettura astratta del prototipo di ambiente per il calcolo parallelo/distribuito implementato secondo la logica Master/Slave oggetto di questa tesi. Verranno discussi il modello Master/Slave, l'architettura astratta del modello ed una possibile implementazione.

3.1 Definizione del modello Master/Slave

Il modello Master/Slave si basa su dispositivo o processo (Master) che controlla uno o più dispositivi o processi (Slave). Il dispositivo o processo Master, di solito, è allocato su un host diverso da dove sono allocati i dispositivi o processi Slave. Una volta stabilita la connessione tra Master e Slave, il Master controlla le operazioni svolte dallo Slave. In conclusione possiamo vedere il modello Master/Slave come un'architettura di rete simile a quella client/server, dove i server sono più di uno), corrispondono agli Slave, rimangono in attesa delle richieste del client; il client, è unico e corrisponde al Master, attivamente effettua le richieste al server e attende anche le risposte.

3.2 Architettura Astratta

In questo paragrafo viene illustrata l'architettura astratta del prototipo di ambiente per il calcolo parallelo/distribuito oggetto di questa tesi implementato con il modello Master/Slave definito nel paragrafo precedente. Inoltre viene illustrato il meccanismo che permette l'iterazione tra il modulo Master ed i moduli Slave. Con l'aggettivo "'astratta"' si vuole indicare che il Master e lo Slave vengono definiti in termini di meccanismi che possono essere implementati sfruttando le funzionalità fornite da differenti strumenti di programmazione. In particolare è possibile utilizzare funzionalita' differenti (fra più linguaggi ma anche all'interno dello stesso) per fornire diverse implementazioni dei meccanismi. Di seguito verranno enumerati tali meccanismi, il loro utilizzo e comportamento e, nel paragrafo successivo, una possibile implementazione. I meccanismi che verranno utilizzati sono:

- *Processi e Thread:*
  un processo è l'unità di concorrenza all'interno di un sistema multitasking [7, 5]. I thread sono flussi di controllo indipendenti all'interno di un dato processo; più thread definiti all'interno dello stesso processo condividono risorse: in particolare il suo spazio di indirizzamento in memoria ed il processore

- *Canale:*
  è un canale di comunicazione biridezionale fra due processi allocati su risorse diverse. Ad esempio se due processi sono allocati su due nodi di elaborazione corrispondenti a due host distinti, il canale permette la comunicazione fra i due processi. Per poter indirizzare un processo è necessario il nome dell'host del nodo di elaborazione su cui è allocato e la porta di sistema sul quale è in ascolto. Si possono individuare due possibili tipi di canale:

  – quello che implementa un meccanismo di affidabilità cioè si assicura che il messaggio inviato verrà ricevuto dal mittente (a patto

che sia raggiungibile). Il protocollo di rete utilizzato è denominato TCP, acronimo di Trasmission Control Protocol

– quello non affidabile perchè la spedizione dei pacchetti (datagram) avviene in modalità best effort e non si garantisce che per ogni pacchtto inviato sia poi ricevuto. Il protocollo di rete corrispondente è denominato UDP, acronimo User Datagram Protocol

come è logico supporre, in condizioni "'paritarie"', il protocollo TCP introduce un overHead di comunicazione sempre maggiore o uguale a quello introdotto da protocollo UDP.

Nella figura 3.1 viene indicata la corrispondenza dei simboli utilizzati nelle figure 3.2 che descrive l'architettura astratta.

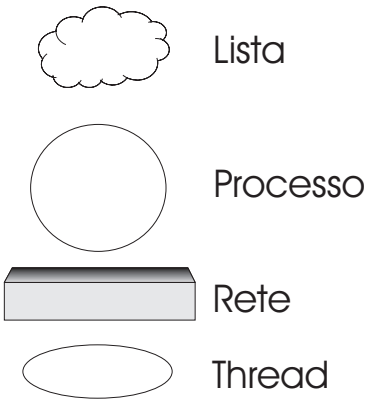

Figura 3.1: simboli che descrivono l'architettura astratta

Il relatore ha proposto questa tesi con lo scopo di effettuare una prima fase di rassegna e di valutare il modello RPC in un contesto simile a quello degli ambienti Litium e Muskel che funzionano in Java RMI.

Per questo motivo viene implementato un prototipo di ambiente di programmazione che supporta computazioni di tipo task farm. Il prototipo è formato da un modulo Master e un modulo Slave. Il modulo Master ha il compito di far eseguire i task su più Slave. Il modulo Slave attende il task

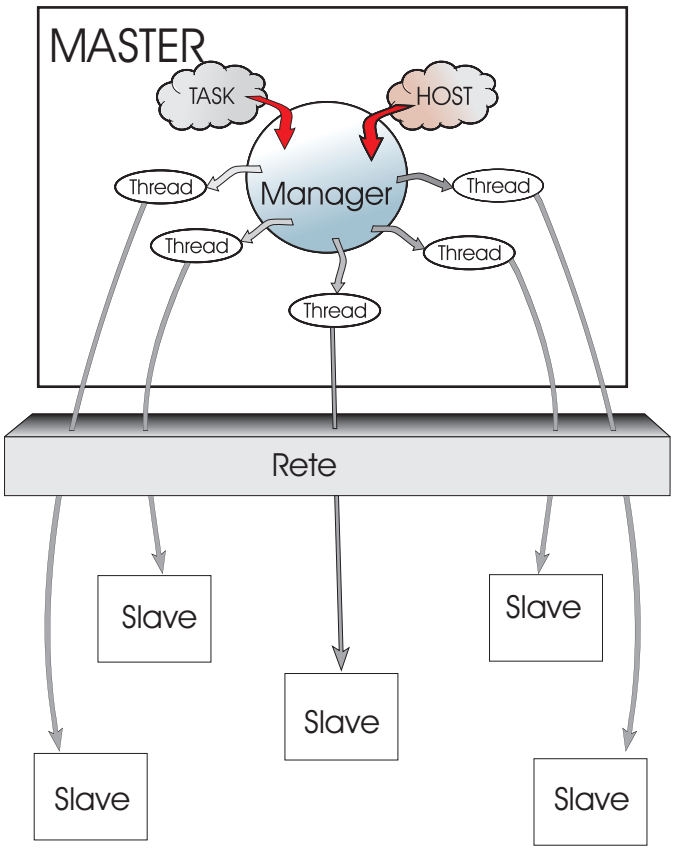

Figura 3.2: architettura astratta

dal modulo Master, una volta ricevuto esegue il calcolo relativo a quel task e restituisce il risultato al modulo Master. Il modulo Master, per inviare i task ai moduli Slave, crea un thread per ogni Slave a cui invia il task. In questo modo il modulo Master implementa auto scheduling (mentre un thread è in attesa attiva di un risultato gli altri thread continueranno la loro esecuzione). Il Master preleva in mutua esclusione i task da una lista. La descrizione del prototipo e visualizzata in figura 3.2.

3.3 Implementazione

In questo paragrafo viene descritta l'implementazione del prototipo di ambiente per il calcolo parallelo/distribuito secondo la logica(il modello) Master/Slave. Per la sua implementazione si è scelto il meccanismo RPC di Sun e il linguaggio di programmazione C. I motivi che hanno portato alla sua scelta sono: esperienza di utilizzo, la sua efficienza, è un linguaggio standard, garantisce la portabilità dei programmi scritti in C (standard, spesso detto ANSI C) su qualsiasi piattaforma, la grammatica e la sintassi sono molto libere e flessibili, utilizza strumenti come che implementano funzionalità a basso livello (puntatori, primitive).

Di seguito si descrive l'implementazione delle entità logiche utilizzate nell'architettura astratta:

- *processi:*
  sono l'entità (eseguibile) derivante dalla compilazione, in particolare i processi implementati nel prototipo oggetto di questa tesi sono di due tipi: il processo che esegue i task ed il processo manager.

- *Thread:*
  è la classica libreria del linguaggio C `pthread.h`. Coincide con le entità a livello logico. Viene utilizzata all'interno del processo manager;

- *Canale:*
  è implementato utilizzando la libreria `rpc.h`. La creazione del canale tra due host avviene in due fasi: nella prima fase l'host chiamante interroga l'altro host di cui conosce l'indirizzo, tramite la `clnt_create(...)`, per verificare se il servizio richiesto è presente su quell'host. Se è presente risponderà inviandogli la porta dove è in ascolto il servizio. Nella seconda fase la comunicazione avviene direttamente tra i due host tramite la `clnt_call(...)`.

3.3.1 Descrizione del prototipo

Il prototipo implementato in questa tesi è composto da una applicazione che come parametri di ingresso riceve uno stream di task da eseguire, e come parametri di uscita restituisce un vettore dei risultati ed il tempo impiegato ad eseguire tutto lo stream di task (tempo di completamento). I task, che formano lo stream di ingresso, per semplicità sono vettori di `double` di lunghezza prefissata (nel capitolo degli Esperimenti vengono effettuate prove con lunghezze diverse di tale vettore). I risultati sono ancora vettori di double della stessa grandezza di quelli di ingresso, per semplicità, il risultato è stampato al video. L'applicazione riceve un task alla volta, e per un determinato numero di iterazioni (ciclo), esegue la funzione applicata ad ogni elemento del vettore contenuto nel task, ed infine inserisce i risultati in un vettore di `double` della stessa dimensione di quello contenuto nel task. L'applicazione si presta ad essere parallelizzata infatti risulta semplice replicare nei vari host la funzione da eseguire, per un determinato numero di volte. Inoltre l'applicazione avra' bisogno di un processo che si occupi di smistare i task ai vari host e gestire i risultati. La funzione non fa altro che calcolare il seno di ogni elemento del vettore contenuto nel task. Nel prototipo implementato in questa tesi si è scelto questa funzione per semplicità anche se non è importante la funzione specifica che viene calcolata ma ai fini dei risultati è importante che la funzione effettui *calcolo*. Inoltre si può osservare che tale applicazione è facilmente adatta al Modello Master/Slave definito all'inizio del capitolo. Infatti è sufficiente implementare il Modulo Master in modo riceva lo stream in ingresso smistare i task ai vari nodi remoti (Slave) ed attendere tutti i risultati, ed implementare il Modulo Slave in modo che rimanga in attesa dei task, li esegua e restituisca il risultato. In dettaglio il modulo:

- Master esegue una prima fase di inizializzazione; genera i thread per l'esecuzione di tutti i task; terminati tutti i thread calcola il tempo di completamento. La fase di inizializzazione consiste nell'inizializzare le strutture dati, prendere gli host da un file, generare i task ed inserirli

in strutture dati appropriate. Nella fase successiva vengono creati un numero prestabilito di thread (variano in base al grado di parallelismo scelto) che come parametro hanno l'host sul quale verranno eseguiti i task, i thread eseguono una chiamata remora verso l'host passatogli come parametro, gli invia il task da eseguire e aspetta (attesa attiva) il risultato, successivamente estrae un nuovo task e lo invia all'host in attesa (ogni thread gestisce un solo host remoto). Questa operazione viene effettuata finchè ci sono task a disposizione dopodichè termina.

- Slave aspetta la richiesta da parte del Master, appena giunge la richiesta di esecuzione del task lo esegue, per un determinato numero di volte, ed restituisce il risultato. Nel caso particolare del prototipo proposto restituisce un vettore di `double`.

Per capire meglio tutta l'architettura è sufficiente vedere la figura 3.2 quella non ancora fatta!

3.3.2 Il modulo Master

In questo paragrafo vengono descritti tutti i dettagli implementativi forniti dal Master.

Le strutture dati

Le strutture dati scelte dove inserire i task e gli host sono liste. Per semplicità di implementazione di queste si è scelto un'unico modello di Lista che andasse bene sia per inserire gli host che i task. Infatti il tipo lista è formato da due campi: `nome` array di caratteri e `arrayInteri` array di `double`. A seconda che venga creata la lista degli host o dei task verrà riempito il campo opportuno. Le strutture dati scelte per inviare i dati attraverso il canale di comunicazione tra Master e Slave sono due: `struct ingresso` che invia (dopo la prima fase di comunicazione) il vettore di interi (task) e il numero ,`NITER`, di iterazioni da eseguire il processo Slave, il quale inserisce nella struttura `struct uscita` il vettore dei risultati e invia tale struttura al Master.

La fase di inizializzazione

La fase di inizializzazione è eseguita dal processo manager il quale esegue:

- l'inizializzazione delle strutture dati per la mutua esclusione, nel caso particolare di questa tesi sono stati usati i semafori presenti nella libreria `semaphore.h`
- vengono creati i task per mezzo della procedura `creaTask(pool_Task)` in particolare crea una lista di `NTASK` elementi (ogni task è un vettore di `DIMENSIONE` elementi)
- preleva gli host dal file passato dall'utente e li inserisce nella lista degli host. Per fare questo sono state fatte delle ipotesi: il file è formato da nomi degli host divisi tra di loro da dei separatori. Come separatori si possono usare quelli di uso piu' comune il nome dell'host ha una lunghezza prefissata.
- infine tramite la procedura `gettimeofday(start,NULL)` inseriamo il tempo in `start` che servira' per calcolare il tempo di completamento.

Gestione dei thread

Dopo la fase di inizializzazione viene fatto un ciclo nel quale ad ogni iterazione vengono settati i parametri da passare al thread (`parametri setParametri(...)` la quale restituisce una struttura dove al suo interno c'è il nome dell'host e il puntatore alla lista dei task), creato un thread (uno per ogni ciclo) e successivamente viene controllato che tutti i thread siano creati. Importante che questa fase sia fatta in mutua esclusione cioè tutti i thread devono essere in esecuzione prima di passare alla fase successiva. Ogni thread esegue una processo `worker` che:

- crea una connessione con il `portmap` dell'host passato come parametro e verifica, interrogando il portmap di tale host, se il processo Slave è in

esecuzione sull'host e in attesa del task. Se è in attesa il portmap dell'host risponde inviando un messaggio contenente anche la porta su cui è in ascolto il processo Slave altrimenti fallisce. La connessione fallisce anche nel caso in cui non riusciamo (per diversi motivi) a connetterci con l'host.

- Se la connessione con il processo Slave sull'host remoto va a buon fine viene estratto, in mutua esclusione, un task (`taskEstratto = get_Mutex(...)`) dalla lista dei task, inviato al processo Slave sull'host remoto dopodichè si attende (con attesa attiva) che il processo sull'host remoto abbia eseguito il task e restituisca il risultato. Infine si continua ad inviare task al processo Slave remoto finchè ci sono task da eseguire. Appena i task termino il processo `worker`.

Fase di terminazione

La fase di terminazione consiste nell' attendere che tutti i thread terminino (`retcode = pthread_join(tid[j], (res[j]))`), effettuando di nuovo una `gettimeofday(stop,NULL)` e applicando la `timeval_subtract (risultato, stop, start)` viene calcolato il tempo di completamento.

Capitolo 4

Esperimenti

In questo capitolo vengono descritte le caratteristiche di efficienza, performance del prototipo, l'ambiente di esecuzione, lo stato dell'ambiente di programmazione, i risultati sperimentali ottenuti. In quest'ultimo paragrafo vengono proposti anche dei grafici e delle tabelle per capire meglio i risultati ottenuti.

4.1 Caratteristiche di efficienza e performance del prototipo

L'obiettivo degli esperimenti effettuati sul prototipo oggetto della tesi è stato quello di verificarne le caratteristiche di efficienza e performance. Per verificare i vari casi viene analizzato il comportamento del prototipo al variare della grana di calcolo. Per variare la grana vengono fatti variare i parametri del prototipo, in particolare variando il tempo speso nel calcolo e il tempo speso nella comunicazione. Per variare il tempo di calcolo del Modulo Slave, in questo prototipo, è sufficiente variare il numero di volte che viene applicata la funzione ad ogni singolo task, in particolare per fare questa modifica basta variare il parametro `NITER`. Per aumentare il tempo speso nelle comunicazioni è sufficiente aumentare la dimensione del vettore di `double` contenuto al

suo interno. Per esempio, nel prototipo implementato in questa tesi basta modificare il valore della costante DIMENSIONE definita nel `file.x`.

4.2 Descrizione dell'ambiente di esecuzione

I test sono stati effettuati utilizzando un cluster di workstation di nome `pianosau`, situato presso il Dipartimento di Informatica dell'Universita degli studi di Pisa [1]. Il cluster è composto da 32 processori pentium III a 800Mhz più un nodo master adibito alle sole funzioni di amministrazione. Tutti i nodi hanno la seguente configurazione: 1 GB di memoria Ram, 2 Hard/disk da 20 GB, 1 Intel(R) Pentium(R) III Mobile CPU 800Mhz, 32 KB cache L1, 3Ethernet Pro 100. I nodi sono connessi tramite tre sottoreti di tipo Fast Ethernet con banda 100Mbit/sec utilizzate nel seguente modo: una abilitata al sistema PVFS [2] al NFS [3] e al traffico di sistema; le restanti due al traffico utente. A seconda della rete utilizzata, i nomi dei nodi del cluster variano da u1 ad u32, da t1 a t32 e da v1 a v32.

4.3 Il prototipo FARM

Il prototipo Master/Slave descritto nel Capitolo 3 implementa una forma di parallelismo di tipo FARM [18, 13]. Infatti come è illustrato nella figura 4.1 è composto dal modulo Master che oltre a generare i task e a gestire i vari host utilizzati per la computazione (i worker), funge anche da emettitore e collettore del FARM. È l'emettitore quando invia i task ai vari worker. È il collettore quando riceve i risultati dai worker. Il Modulo Slave invece è composto semplicemente dal worker. Lo stream di input è costituito dai task da eseguire. I task del prototipo sono generati ed inseriti in una struttura a lista. L'emettitore (un thread dell'emettitore, in verità) si connette ad un worker, e in un ciclo invia al worker uno dei task che appaiono sullo stream

[1] le informazione sono state ricavate dal sito http://pianosa.di.unipi.it

[2] PVFS: Paralel Virtual File Sistem

[3] NFS: Network File System

di ingresso (cioè nella lista), attende che il task venga valutato, quindi riceve il risultato lo tratta (ovvero nel nostro caso lo stampa al video) e passa ad una nuova iterazione.

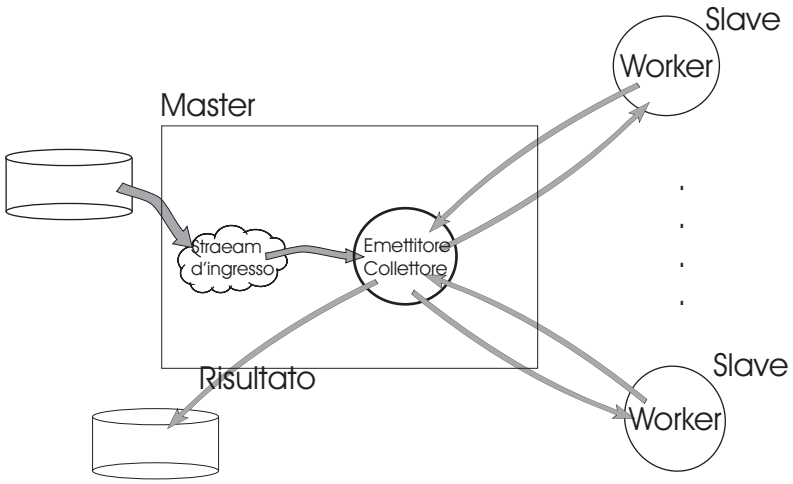

Figura 4.1: Rappresentazione del Farm.

4.4 Descrizione dello stato dell'ambiente di programmazione

Le prove sono state effettuate utilizzando il cluster pianosa ed utilizzando fino a 24 nodi dei 32 disponibili. Non si è potuto sfruttare tutti i 32 nodi dato che oltre i 24 nodi si passa ad un altro chassis (backplane) non si è ritenuto conveniente utilizzare anche questi altri nodi visto che le comunicazioni avrebbero avuto costi diversi. Tutti i test sono stati effettuati in modo che il carico sul processore fosse nullo: questo affinché il tempo di servizio dei singoli slave non fosse falsato da carichi dovuti ad altri utenti, in modo da poter valutare correttamente l'efficienza. Inoltre nel caso in cui, durante l'esecuzione, avviene il fallimento di un nodo, si prosegue l'esecuzione e solo al termine dell'esecuzione viene segnalato un errore.

4.5 Risultati

I risultati ottenuti dagli esperimenti svolti nelle diverse situazione sono stati inseriti nelle tabelle di figura 4.2 e 4.3.

DIMENSIONE DEL TASK INVIATO	TEMPO DI ESECUZIONE DI UN TASK (in msec)	TEMPO DI COMPLETAMENTO DI 1024 TASK IN BASE AL GRADO DI PARALLELISMO (in SEC)				
		2	4	8	16	24
16 byte	0,115	0,7108	0,4169	0,2201	0,0814	0,2679
16 byte	102,19	57,9696	28,9641	14,5367	7,3098	4,8981
16 byte	204,311	114,9732	57,4397	28,8309	14,5391	9,6501
16 byte	1663,515	913,3325	456,5408	228,3485	114,0932	76,6276
32 byte	0,116	0,7247	0,3772	0,193495	0,0883	0,0679
32 byte	104,526	57,0551	28,6553	14,4315	7,1941	4,7787
32 byte	208,541	113,2648	56,7593	28,4541	14,3389	9,5066
32 byte	1675,772	899,5882	449,9807	224,9197	112,5546	75,4953
2 kbyte	0,241	2,5087	1,2412	1,4338	1,3515	3,0251
2 kbyte	112,635	58,2574	29,2505	14,8052	7,7138	5,1201
2 kbyte	216,679	114,2269	57,1468	28,7791	14,5865	9,8122
2 kbyte	1711,415	886,9609	443,858	221,8053	111,0236	74,7051

Figura 4.2: Tabella contenete i tempi di completamento nei diversi casi.

La tabella di figura 4.2 contiene le seguenti colonne:

- ***Dimensione del task inviato:*** le diverse grandezze dei task inviati tra il Modulo Master e il MOdulo Slave misurate in byte;
- ***Tempo di esecuzione di un task:*** contiene il tempo di esecuzione di un singolo task misurato all'interno del Modulo Slave. Il tempo è misurato in millisecondi;
- ***Tempo di completamento per 1024 task in base al grado di parallelismo:*** in queste colonne troviamo i valori dei tempi di completamento al variare del grado di parallelismo misurate in secondi;

La tabella di figura 4.3 contiene le seguenti colonne:

- ***Dimensione del task inviato:*** è la stessa della tabella di figura 4.2

- ***Tempo di esecuzione di un singolo task:*** è la stessa della tabella di figura 4.2

- ***Efficienza in base al grado di parallelismo:*** contiene i valori di efficienza calcolati al variare del grado di parallelismo, grandezza del task, tempo di esecuzione del singolo task. Per calcolare l'efficienza è stato misurato il tempo di completamento, nei diversi casi, nel caso sequenziale ovvero l'esecuzione di un programma equivalente al solo modulo Slave (senza Master)

I risultati ottenuti nella tabella di figura 4.2 si possono dividere sostanzialmente in 3 categorie. In ognuna delle categorie abbiamo una differente grana [18]. In particolare, la grana è:

DIMENSIONE DEL TASK INVIATO	TEMPO DI ESECUZIONE DI UN TASK *IN MILLISECONDI*	EFFICIENZA IN BASE AL GRADO DI PARALLELISMO				
		2	4	8	16	24
16 byte	0,115	0,077307259	0,065903094	0,062414811	0,084382678	0,084382678
16 byte	102,19	0,900644476	0,901288146	0,897899798	0,892808285	0,888269601
16 byte	204,311	0,908168164	0,908909691	0,90540878	0,897708593	0,901674594
16 byte	1663,515	0,914579302	0,914830613	0,914517722	0,915167381	0,908413983
32 byte	0,116	0,080792052	0,077611347	0,07564795	0,082885051	0,071858125
32 byte	104,526	0,941545979	0,937348414	0,930603194	0,933403761	0,936795921
32 byte	208,541	0,948529464	0,946408782	0,943932509	0,936569402	0,941758007
32 byte	1675,772	0,955398259	0,95500207	0,955302048	0,954497861	0,948695923
2 kbyte	0,241	0,052577032	0,053134064	0,022998326	0,012199408	0,003633489
2 kbyte	112,635	0,955415106	0,951436728	0,939872477	0,901954938	0,905906786
2 kbyte	216,679	0,971531224	0,970964253	0,96402424	0,951007781	0,94249166
2 kbyte	1711,415	0,987935319	0,987094972	0,987645471	0,986569522	0,977465617

Figura 4.3: Tabella contenete l'efficienza nei diversi casi.

1. BASSA siamo nel caso in cui il tempo impiegato nelle comunicazioni diventa rilevante ai fini del calcolo del tempo di completamento. In questi casi come si vede dalla tabella di figura 4.2 il tempo di esecuzione di un singolo task da parte dello Slave varia da 0,115 msec a 0,330 msec quindi rispetto al tempo necessario per comunicare i dati del task al

worker remoto che vale di circa 0,0725 msec nel caso in cui il task e 16byte, 0,077 nel caso in cui il task e 32byte, infine 0,877 nel caso in cui il task è grande 2kbyte. Quindi il tempo di esecuzione del singolo task è paragonabile al tempo impiegato nelle comunicazioni addirittura nell'ultimo caso e maggiore il tempo speso nella comunicazione rispetto a quello speso nel calcolo. In questi casi come si vede nei grafici in figura 4.4, 4.5, 4.6 all'aumentare del grado di parallelismo l'efficienza è bassa (inferiore a 0,5) e aumenta poco il suo valor,oppure, in alcuni casi, il suo valore diminusce. Allo stesso modo il comportamento del tempo di completamento diminuisce di poco e nei casi peggiori aumenta all'aumentare del grado di parallelismo.

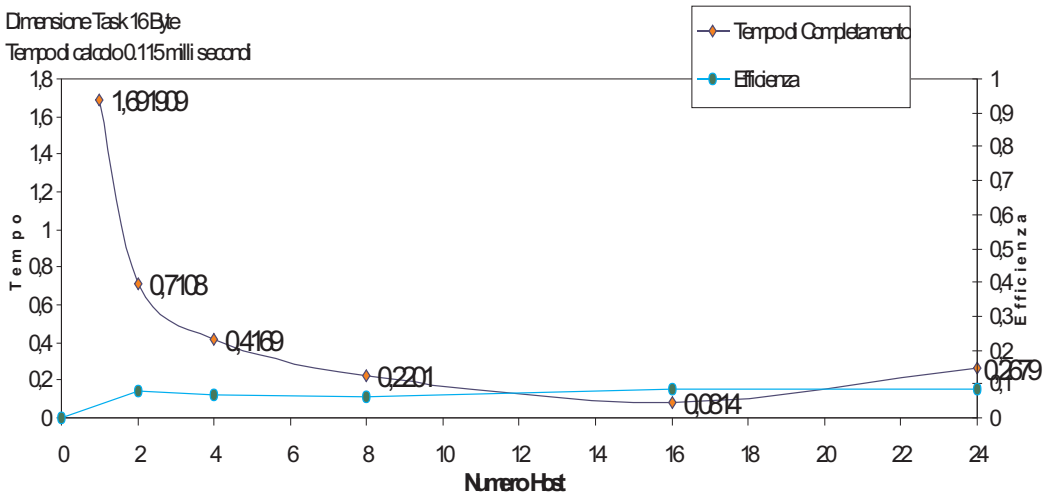

Figura 4.4: grana bassa, pochi dati

2. MEDIA siamo nel caso in cui il tempo impiegato nelle comunicazioni diventa poco rilevante ai fini del calcolo del tempo di completamento. Dalla tabella di figura 4.2 notiamo che in alcune righe il tempo di calcolo di un singolo task da parte dello slave varia da 100 millisec a 200millisec circa. Quindi il tempo in cui lo Slave svolge effettivamente del *calcolo* è di ben tre ordini di grandezza superiore al caso precedente (con grana BASSA). Il tempo di comunicazione varia sempre da

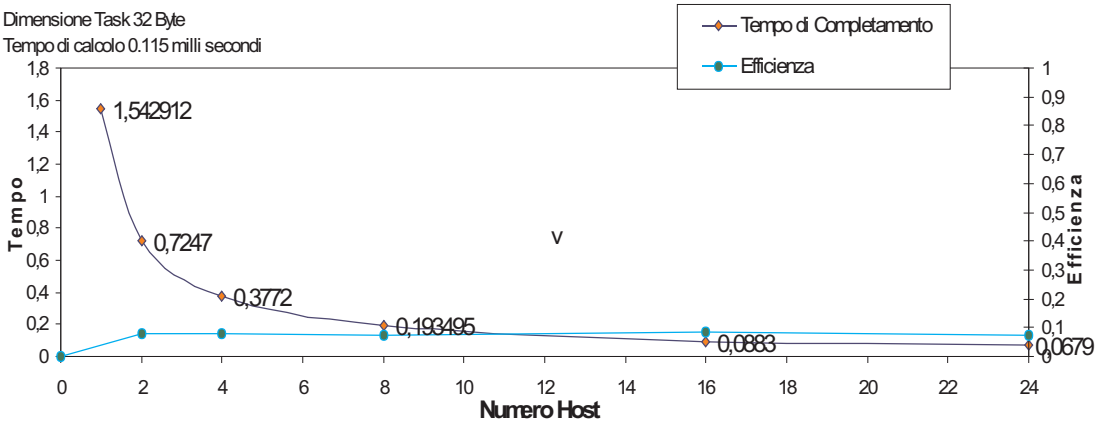

Figura 4.5: grana bassa, pochi dati

0,0725 millisec a 0,877 (la grandezza dei dati inviati è la stessa) per cui la differenza tra tempo di calcolo, che varia da 100 a 200 millisec, e il tempo di comunicazione è molto meno rilevante. Infatti come si vede nei grafici di figura4.7, 4.8, 4.9, all'aumentare del grado di parallelismo l'efficienza tende ad avere valori in torno allo 0,9 che inizia ad essere un buon risultato. I grafici mostrano anche come il tempo di completamento all'aumentare del grado di parallelismo diminuisce con l'andamento desiderato. Anche nei seguenti grafici di figura 4.10, 4.11, 4.12, i valori dell'efficienza e del tempo di completamento sono buoni, leggermente migliori dei grafici appena mostrati dato che il tempo di *calcolo* è solo raddoppiato.

3. MEDIO ALTA In quest'ultimo caso il tempo di esecuzione di un task da parte dello Slave raggiunge i 1700 millisecondi (ovvero circa dell'ordine del secondo). Dato che i valori dei tempi di comunicazione sono gli stessi che nei punti precedenti (la dimensione dei task inviati nei vari casi è la stessa) è del tutto irrilevante rispetto al tempo di calcolo da parte del Master. Infatti nei grafici di figura 4.10, 4.11, 4.12, il tempo di completamento ha l'andamento desiderato (scala), l'efficienza è ottima infatti valori superano 0.9 e tendono ad avere valori vicini a quelli ideali.

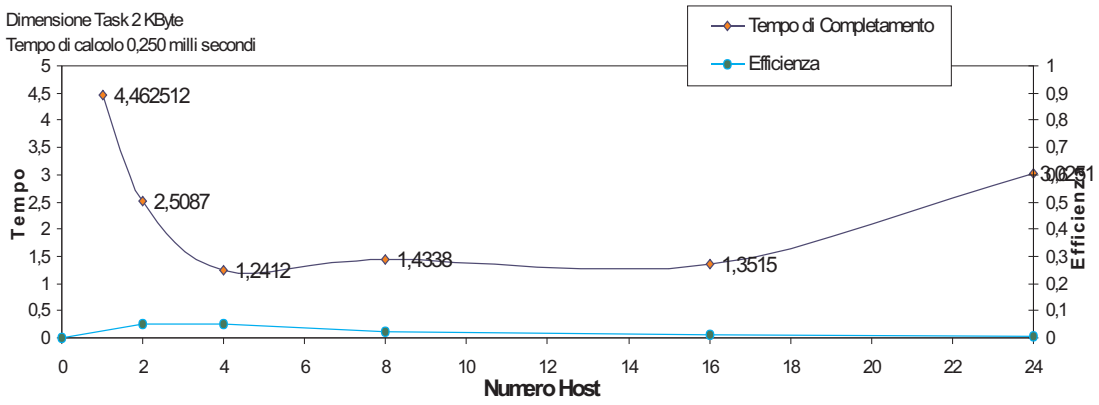

Figura 4.6: grana bassa, dati piu' consistenti

I risultati ottenuti hanno rispettato le aspettative prefissate. Infatti nei casi in cui lo slave lavora poco (nell'ordine dei decimi di millisecondo) si riscontrano una serie di problemi: il tempo impiegato nelle comunicazioni tra modulo Master e modulo Slave è dello stesso ordine (se non maggiore) del tempo di calcolo del modulo Slave stesso; se il modulo Slave lavora poco, il modulo Master, che si occupa di inviare i task ai moduli Slave, è il collo di bottiglia dato che deve inviare gli slave velocemente ed i thread in escuzione al suo interno (il loro numero varia in base al grado di parallelismo) devono accedere alla lista dei task, da inviare allo slave, in mutua esclusione, quindi la probabilità che un thread debba attendere prima di accedere alla suddetta lista (perché in quell'istante un altro thread sta prelevando un task) è abbastanza alta (vedere risultati grafici di figura 4.4, 4.5, 4.6) e questo rallenta ancora di più il sistema e le sue prestazioni.

Negli altri casi, aumentando il tempo di calcolo dello slave, il tempo di comunicazione ha influenzato molto meno e in alcuni casi quasi per niente l'efficienza che è passata da valori buoni (intorno allo 0,9) fino ad avere risultati che si avvicinano molto all'ottimo (valori intorno allo 0.99). Il caso in cui ci si è avvicinati di più all'ottimo è rappresentato nel grafico 4.15. Non è stato un risultato isolato, infatti, nei grafici di figura 4.12 e 4.11 l'efficienza

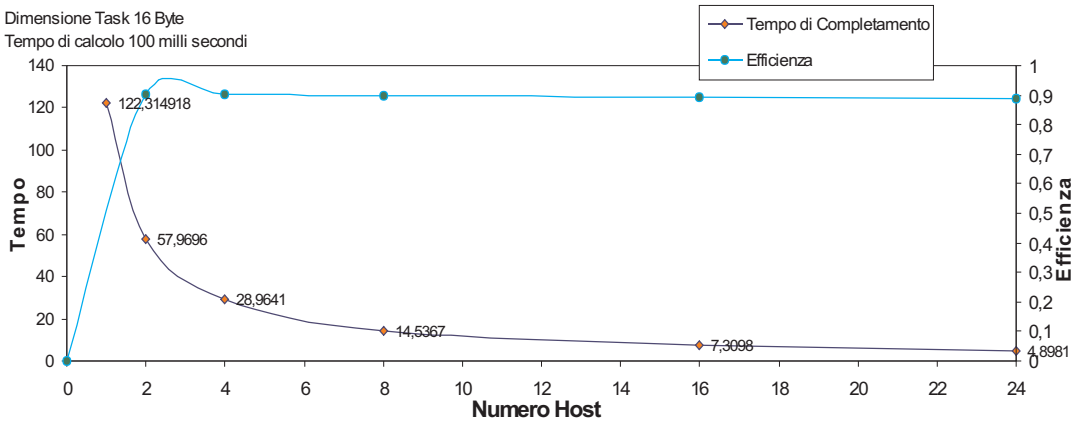

Figura 4.7: grana media, pochi dati

supera il valore di 0.95.

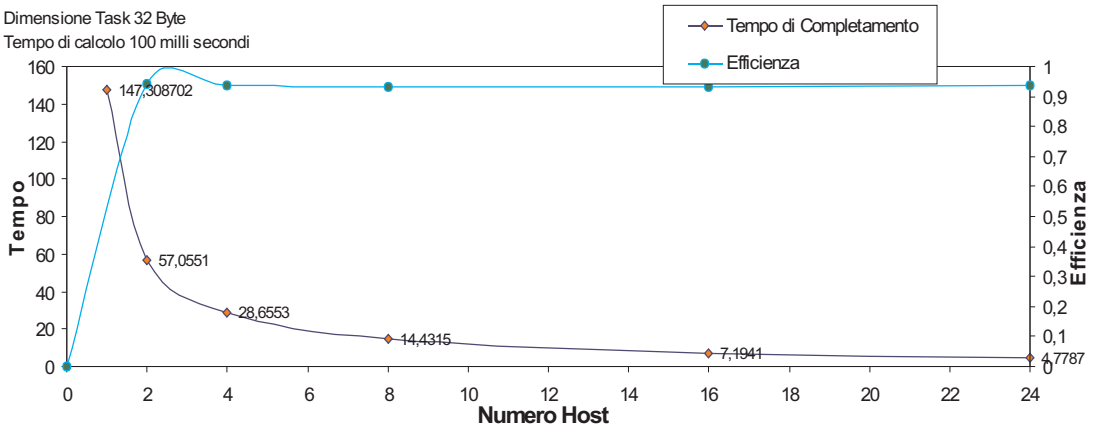

Figura 4.8: grana media, pochi dati

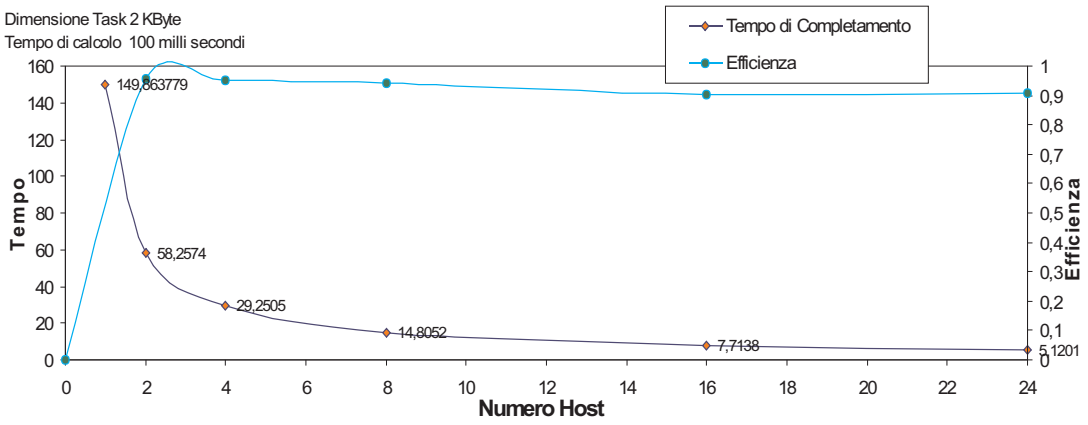

Figura 4.9: grana media, dati consistenti

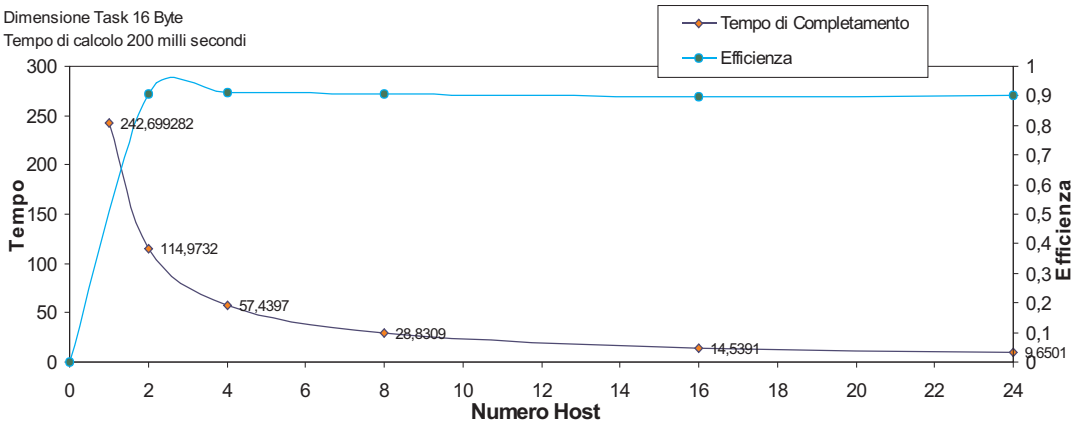

Figura 4.10: grana media, pochi dati

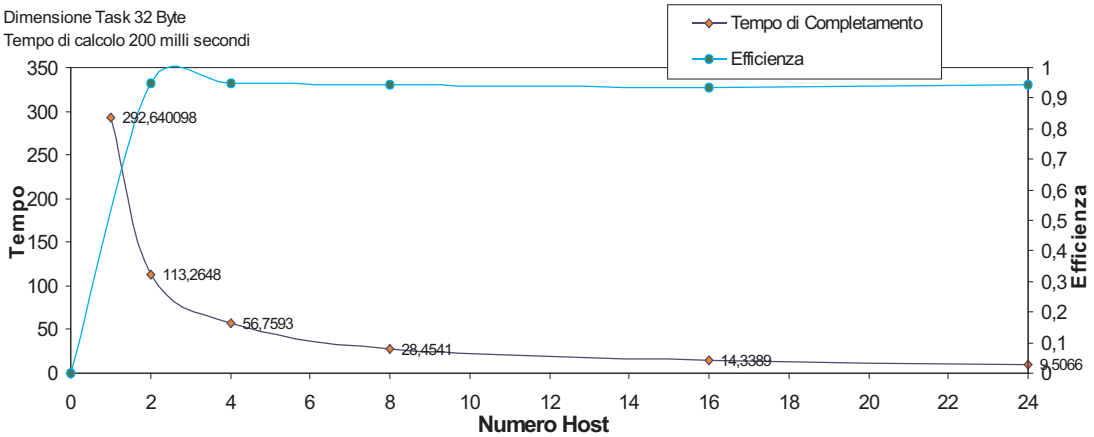

Figura 4.11: grana media, pochi dati

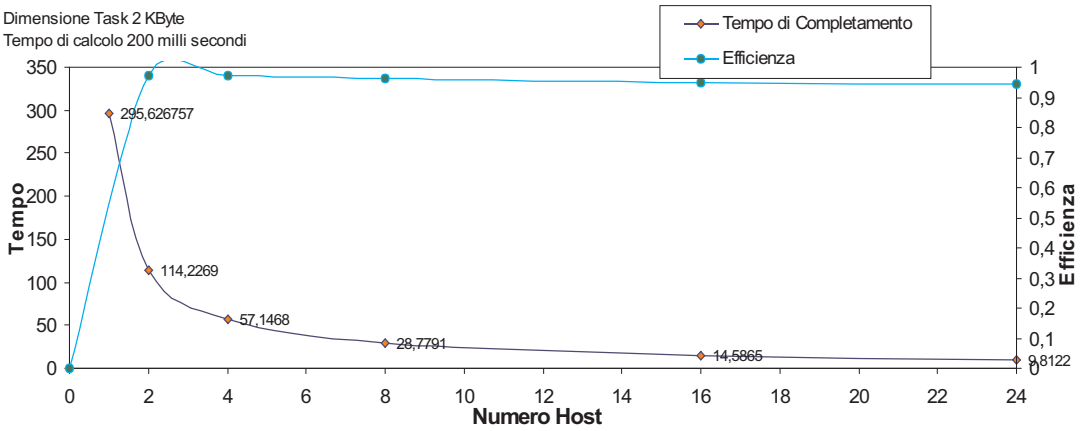

Figura 4.12: grana media, dati consistenti

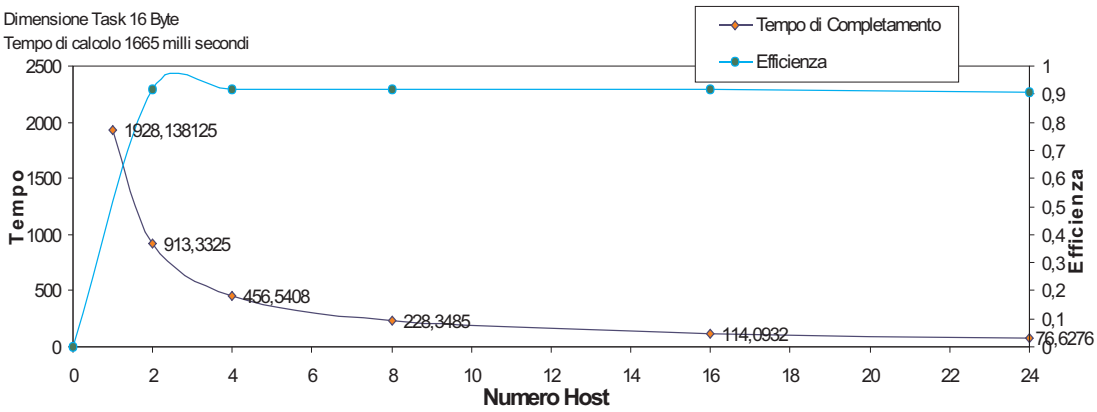

Figura 4.13: grana medio alta, pochi dati

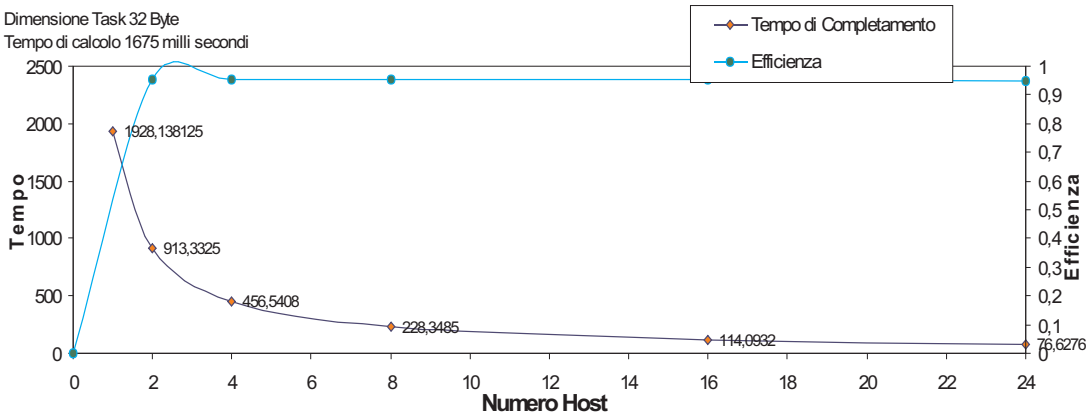

Figura 4.14: grana medio alta, pochi dati

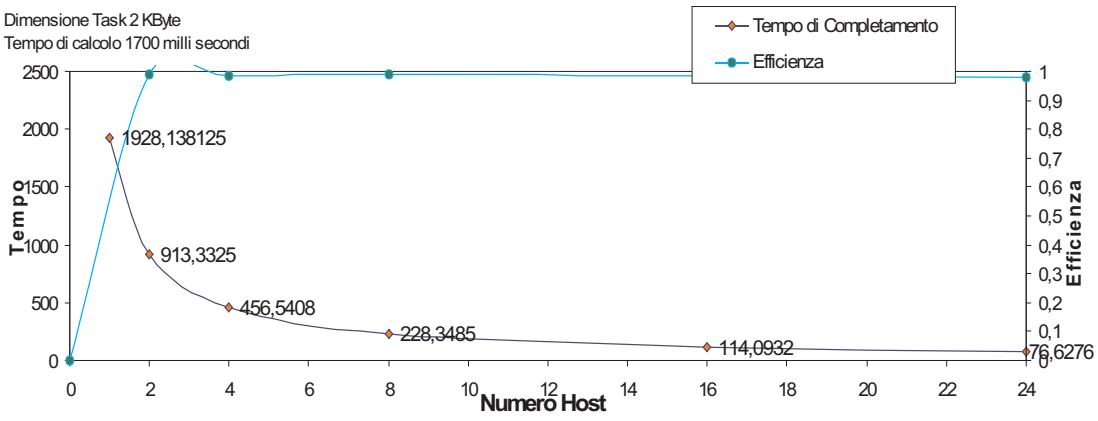

Figura 4.15: grana medio alta, dati consistenti

Capitolo 5

Conclusioni

Questa tesi è stata sostanzialmente organizzata in due parti: una rassegna dei meccanismi e tecniche della chiamata di procedura/metodo remota/o, in particolare RPC di Sun Microsystem e Java RMI, e l'implementazione di un semplice prototipo di ambiente di programmazione strutturato che utilizza i meccanismi di RPC. La rassegna è concentrata sulla definizione e le differenze dei due meccanismi, RPC e RMI, e successivamente su una breve introduzione a CORBA. Lo scopo finale della tesi é stato quello di valutare il modello RPC in un contesto simile a quello degli ambienti Litium e Muskel [4] che funzionano in Java RMI. Per questo motivo, nella seconda parte della tesi è stato implementato un prototipo di ambiente per il calcolo parallelo distribuito secondo la logica (il modello) Master/Slave adottata in Muskel.

L'applicazione (cioè il prototipo di ambiente di programmazione che supporta computazioni di tipo task farm) è stata sviluppata utilizzando l'ambiente di programmazione C in modo da ottenere una buona efficienza (scalabilità). In particolare, si è utilizzato l'ambiente RPC sotto linux, derivato da quello classico Sun, al fine di sfruttare la rete e fare calcolo distribuito.

L'applicazione si divide di fatto in due moduli: il Modulo Master ed il Modulo Slave. Il Modulo Master si occupa di far eseguire i task ai diversi Slave, in esecuzione su diversi host, appena gli Slave hanno terminato l'esecuzione di tutti i task il Modulo Master termina.

Il Modulo Slave attende che il Modulo Master gli invii un task da eseguire. Appena riceve il task esegue il calcolo relativo a quel task e restituisce il risultato della computazione al modulo Master.

Le comunicazioni tra modulo Master e modulo Slave avvengono attraverso lo standard RPC, quindi sfruttando il codice ottenuto mediante la compilazione del file con estensione `.x` con il compilatore `rpcgen`.

Per gestire in modo concorrente l'invio dei Task ai diversi Moduli Slave, in esecuzione su nodi diversi, l'implementazione del modulo Master sfrutta tecniche multithreading standard.

Per questo motivo il Modulo Master ha come parametro di ingresso, oltre alla lista dei task, la lista dei nomi di tutti gli host sui quali inviare i task da eseguire. Il Modulo Master invia un nuovo task da eseguire all'i-esimo Slave solo nell'istante successivo alla ricezione del risultato giunto proprio dell'i-esimo Slave relativo all'elaboratore del task precedente.

Nel Capitolo 5 sono stati mostrati i risultati ottenuti dall'esecuzione del prototipo Master/Slave facendo variare la grana del calcolo e grado di parallelismo. Per tutte queste prove è stato calcolato il tempo di completamento e calcolata l'efficienza dell'applicazione.

Si è osservato, anche tramite l'uso dei grafici presentati nel Capitolo 4, come per grane di elaborazione medio/alte l'efficienza osservata risulta molto vicina a quella ideale.

Per ragioni di tempo, la tesi si è conclusa con la verifica sperimentale della fattibilità della realizzazione di un supporto per programmazione parallela strutturata che implementi il paradigma task farm mediante i meccanismi tipici RPC presenti in modo nativo nel sistema operativo Linux. Questi risultati potrebbero essere confrontati direttamente con quelli ottenuti utilizzando il prototipi Muskel. Per fare questo, visto che Muskel è implementato completamente in Java, occorrerebbe avere a disposizione task che richiedano approssimativamente la stessa quantità di calcolo e la stessa quantità di dati in ingresso/uscita dei loro corrispettivi utilizzati nel prototipo C/RPC realizzato nel corso della tesi. Utilizzando questi task si potrebbe confrontare

direttamente l'efficienza dei due prototipi per task di grana fine/media/alta al fine di verificare quanto overhead introduce la macchina virtuale Java.

Adottando nel prototipo le stesse caratteristiche già implementate nel prototipo Muskel, ovvero la possibilità di implementare computazioni parallele strutturate a skeleton con composizioni arbitrarie di pipeline, farm e task sequentziali, si potrebbe inoltre facilmente estendere il prototipo in modo da permettere il calcolo di programmi paralleli strutturati più complessi del semplice caso task farm qui considerato.

Inoltre, sarebbe da considerare la possibilità di utilizzare il meccanismo RPC, come considerato in questa tesi per l'implementazione di computazioni parallel di tipo task farm, in modalità *sicura*, ovvero utilizzando protocolli tipo SSL, al fine di garantire che i dati trasmessi per l'elaborazione distribuita non possano essere intercettati, letti e/o modificati, come peraltro già studiato per Muskel [3].

Bibliografia

[1] A brief tutorial on corba, (sito web). *http://www.cs.indiana.edu/ kksiazek/tuto.html*.

[2] Document object model (dom), (sito web). *http://www.w3.org/DOM/*.

[3] M.aldinucci e m.danelutto. the cost of security in skeletal systems. *Tecnical Report: TR-06-03*.

[4] Muskel, (sito web.

[5] Yolinux tutorial: Posix thread (pthread) libraries, (sito web). *http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html*.

[6] Manuale remote procedure call. *http://www.sun.com*, 1990.

[7] Tutorial thread, (sito web). *http://www.cs.cf.ac.uk/Dave/C/node29.html*, 1999.

[8] Overview of corba, (sito web). *http://www.cs.wustl.edu/ schmidt/corba-overview.html*, 2004.

[9] le classi java, (sito web). *http://java.sun.com/j2se/1.5.0/docs/api/index.html ?java/lang/Object.html*, 2006.

[10] Omg, (sito web). *http://www.omg.org*, 2006.

[11] Ira Pohl Al Kelly. *Didattica e Programmazione C*. ADDISON - WESLEY.

[12] Dennis M.Ritchie Brian W.Kernighan. *LINGUAGGIO C.* Gruppo Editoriale Jackson, 1985.

[13] Marco Danelutto. *Sistemi Operativi 2.* Srvizio Editoriale Universitario di Pisa, 2001.

[14] Jean Dollimore e Tim Kindberg George Coulouris. Archivie material from edition 2 of distribuited system: Concepts and design. *http://www.cdk3.net/rmi/Ed2/SunRPC.pdf*, 1994.

[15] OMG Group. Corba basics, (sito web). *http://www.omg.org/gettingstarted/corbafaq.htm.*

[16] Elliotte Rusty Harold. *Java Network Programming.* O'Reilly e Associates, 2004.

[17] (sito web) Sun Developer Network (SDN). Java remote method invocation / distributed computing for java. *http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html.*

[18] Marco Vanneschi. *Architettura degli Elaboratori II.* Servizio Editoriale Universitario di Pisa, 2000.

Appendice A

AppendiceA

A.1 Introduzione

Nell'appendiceA viene introdotto il codice con cui è stato implementato il prototipo oggetto di questa tesi. Il codice introdotto è diviso in più file, tra questi vengono inseriti sia quelli generati dalla compilazione del file `array.x` che quelli scritti sostanzialmente per: fare operazioni sui file; effettuare operazioni sulle liste; oltre che a quelli più importanti che implementano il modulo Master e il Modulo Slave. Nei seguenti capitoli viene inserito il codice di tutti questi file in ordine alfabetico con una piccola descrizione delle funzionalità svolte da ogni file.

A.2 Codice del prototipo oggetto della tesi:

A.2.1 array.x

`array.x` è un file scritto in un linguaggio simile al C (l'Interface Definition Language). Questo file permette la generazione, attraverso la compilazione con `rpcgen`, dei file che servono ad interfacciare il client ed il server. Il file `array.x` contiene il nome del programma, la descrizione delle procedure, la descrizione delle strutture dati da pubblicare come RPC.

```
const DIMENSIONE = 1024;
const NITER = 16000000;
const LUNGHEZZAHOST = 100;
const NTASK = 1024;
struct ingresso { double vectorIN[DIMENSIONE];
                  int Niter;
                };
struct uscita { double vectorOUT[DIMENSIONE];
                int finale;
              };
program ARRAY_PROG{
  version ARRAY_VERSION{
    struct uscita consumatask(struct ingresso) = 1;
  } = 1;
} = 0x20000007;
```

I file generati dalla compilazione del `array.x` sono: `array_clnt.c`, `array_svc.c`, `array_xdr.c`, `array.h`.

A.2.2 array_clnt.c

Il file `array_clnt.c` contiene la procedura `struct uscita * consumatask_1( struct ingresso *argp, CLIENT *clnt)`. Questa attraverso la `clnt_call(...)` chiama la procedura remota `consumatask` associata all'handle `clnt` ottenuto tramite la chiamata RPC `clnt_create(...)`. Questa procedura come parametro di uscita restituisce una variabile statica. Questo inserito in un contesto che utilizza i thread può avere conseguenze disastrose sul risultato di un generico thread (nel caso del modello Master/Slave implementato). Tale risultato può essere sovrascritto restituendo così un valore sbagliato e senza che si verifichi nessun tipo di errore. Per questo motivo è stato modificato questo file, praticamente è stato scritto di nuovo questo file che esegue la stessa cosa ma in maniera opportuna cioè è fatto in modo che da non restituire una variabile statica come risultato. Il nuovo file ha il seguente codice:

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include <memory.h> /* for memset */
#include "array.h"
/* Default timeout can be changed using clnt\_control() */
static struct timeval TIMEOUT = { 25, 0 };
struct uscita *
consumatask_1(struct ingresso *argp, CLIENT *clnt)
{
        static struct uscita clnt_res;
        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, consumatask,
                (xdrproc_t) xdr_ingresso, (caddr_t) argp,
                (xdrproc_t) xdr_uscita, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
```

```
        }
        return (&clnt_res);
}
```

il file che sostituisce `array_clnt.c` non restituisce più una struttura di tipo static. Tale file è riportato di seguito:

```
 /* MODIFICA DEL FILE array_clnt.c */
#include <memory.h> /* for memset */
#include "array.h"
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };
/* Restituisce 0 in caso di successo,
                              nonzero in caso di errore */
int consumatask_bella(struct ingresso *arg,
                      struct uscita *risultato,
                      CLIENT *clnt){
  if (clnt_call (clnt,
                 consumatask,
                 (xdrproc_t) xdr_ingresso,
                 (caddr_t) arg,
                 (xdrproc_t) xdr_uscita,
                 (caddr_t) risultato,
                 TIMEOUT)
      != RPC_SUCCESS)
    return -1;
  else
    return 0;
}
```

A.2.3 array_svc.c

contiene il `main` che viene eseguito sullo Slave dove al suo interno crea un socket, sul quale riceve le richieste da eseguire, si registra sul portmap dell'host su cui viene eseguito e per finire si mette in attesa di ricevere le richieste da parte del Modulo Master.

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include "array.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif
static void
array_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        union {
                struct ingresso consumatask_1_arg;
        } argument;
        char *result;
        xdrproc_t _xdr_argument, _xdr_result;
        char *(*local)(char *, struct svc_req *);
        switch (rqstp->rq_proc) {
        case NULLPROC:
```

```
(void)svc_sendreply(transp,(xdrproc_t)xdr_void,(char *)NULL);
        return;
case consumatask:
    _xdr_argument = (xdrproc_t) xdr_ingresso;
    _xdr_result = (xdrproc_t) xdr_uscita;
local=(char *(*)(char *,struct svc_req *))consumatask_1_svc;
                break;
        default:
                svcerr_noproc (transp);
                return;
        }
        memset ((char *)&argument, 0, sizeof (argument));
        if (!svc_getargs (transp, (xdrproc_t) _xdr_argument,
                                        (caddr_t) &argument)){
                svcerr_decode (transp);
                return;
        }
        result = (*local)((char *)&argument, rqstp);
        if (result != NULL && !svc_sendreply(transp,
                        (xdrproc_t) _xdr_result, result)){
                svcerr_systemerr (transp);
        }
        if (!svc_freeargs(transp,(xdrproc_t)_xdr_argument,
                                        (caddr_t) &argument)){
            fprintf(stderr,"%s","unable to free arguments");
            exit (1);
        }
        return;
}
int
main (int argc, char **argv)
```

```
{
  register SVCXPRT *transp;
  pmap_unset (ARRAY_PROG, ARRAY_VERSION);
  transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL) {
    fprintf(stderr,"%s","cannot create udp service.");
    exit(1);
  }
  if (!svc_register(transp, ARRAY_PROG,
           ARRAY_VERSION,array_prog_1, IPPROTO_UDP)) {
    fprintf (stderr, "%s", "unable to register
                  (ARRAY_PROG, ARRAY_VERSION, udp).");
    exit(1);
                              }
  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {
    fprintf(stderr,"%s","cannot create tcp service.");
    exit(1);
  }
  if (!svc_register(transp, ARRAY_PROG, ARRAY_VERSION,
                         array_prog_1, IPPROTO_TCP)) {
    fprintf (stderr, "%s", "unable to register
                  (ARRAY_PROG, ARRAY_VERSION, tcp).");
    exit(1);
  }
  svc_run ();
  fprintf (stderr, "%s", "svc_run returned");
  exit (1);
  /* NOTREACHED */
}
```

A.2.4 array_xdr.c

Questo file viene utilizzato per tradurre le strutture dati presenti nel `array.x` in un formato xdr ovvero indipendente dalla macchina su cui viene eseguito il codice di questo prototipo.

```
*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "array.h"

bool_t
xdr_ingresso (XDR *xdrs, ingresso *objp)
{
   register int32_t *buf;
   int i;
   if (!xdr_vector (xdrs, (char *)objp->vectorIN, DIMENSIONE,
            sizeof (double), (xdrproc_t) xdr_double))
            return FALSE;
   if (!xdr_int (xdrs, &objp->Niter))
             return FALSE;
   return TRUE;
}

bool_t
xdr_uscita (XDR *xdrs, uscita *objp)
{
    register int32_t *buf;
    int i;
    if (!xdr_vector (xdrs, (char *)objp->vectorOUT, DIMENSIONE,
            sizeof (double), (xdrproc_t) xdr_double))
```

```
            return FALSE;
    if (!xdr_int (xdrs, &objp->finale))
            return FALSE;
    return TRUE;
}
```

A.2.5 error.c

Il file `error.c` contiene le procedure utilizzate per controllare gli eventuali errori durante l'esecuzione del programma.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "lista.h"
void * ERROR_1KILL(int retcode, char* messaggio){
  if(retcode == -1){
    perror(messaggio); exit(errno);
  }else{return (void *)retcode;}
}
void * ERROR_1(int retcode, char* messaggio){
  if(retcode == -1){
    perror(messaggio); return (void *)errno;
  }else{return (void *)retcode;}
}
void * ERROR_0(int retcode, char* messaggio){
  if(retcode == 0){ perror(messaggio); return (void *)errno;
  }else{return (void *)retcode;}
}
void * ERROR_NULL(lista *list ,char *messaggio){
  if(list == NULL){ perror(messaggio);return NULL;
  }else{return (void *)list;}
}
```

A.2.6 file.c

Il `file.c` contiene le procedure necessarie ad eseguire operazioni sui file. In particolare contiene la procedura `memorizzaFile(TC_lista *listaHost, char *nomeFile)` che viene utilizzata per inserire i nomi degli host contenuti nel `nomeFile` nella `listaHost`. Tale procedura scansiona tutto il file leggendone un carattere alla volta. I caratteri letti vengono memorizzati in un vettore temporaneo se sono lettere dell'alfabeto, se invece sono caratteri speciali (ad esempio "spazio", "return" etc.) significa che il nome dell'host è terminato quindi viene copiato il vettore temporaneo nella `listaHost`. Questa operazione viene compiuta finché non viene raggiunta la fine del file. Per riconoscere se il carattere letto dalla `memorizzaFile(TC_lista *listaHost, char *nomeFile)` è un carattere oppure un separatore viene utilizzata la procedura `e_un_separatore(char c)`.

Inoltre contiene le procedure `leggiFile(char nomeFile)` che legge un file di `double`, `scriviFile(char *nomeFile, double *tempo)` che scrive il contenuto della variabile `tempo` all'interno del `nomeFile`. Per finire contiene la procedura `void *calcolaMedia(char *nomeFile)` che esegue la media dei valori contenuti nel `nomeFile`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include<assert.h>
#include <unistd.h>
#include <math.h>
#include "file.h"
#include"array.h"
#include"error.h"
int e_un_separatore(char c){
  switch(c){
  case ' ': case '\n': case '\r': case '\t': case '\0':
    return 1;
```

```
  default:
    return 0;
  }
}
void *memorizzaFile(TC_lista *listaHost, char *nomeFile){
 enum stato stato = stato_separatore;
 int k = 0;/*indice del vector dove metto l'host*/
 char vector[LUNGHEZZAHOST];/*per ora lo dichiaro statico*/
 char leggi;
 int fd,j;
 /*apro il file in lettura*/
 fd = open(nomeFile, O_RDONLY);
  ERROR_1KILL(fd,"aprendo il file\n");
 do{ /*leggo un carattere dal file*/
   j = read(fd, &leggi, sizeof(char));
   ERROR_1(j,"leggendo il file\n");
   switch(stato){
   case stato_separatore:{
     if(e_un_separatore(leggi)){
       /* Non si fa nulla */
     }
     else{ /* Inizio a trattare un nuovo nome,
          di cui conosco il primo carattere:*/
       vector[k] = leggi;
       stato = stato_roba;
       k++;
     }
     break;
   }
   case stato_roba:{
     if(e\_un\_separatore(leggi)){
```

```
      /*cambio di stato dato che ho trovato l'host*/
      stato = stato_separatore;
      /*nome host finito metto il terminatore*/
      vector[k] = '\0';
      k =0 ;
      inserisci(listaHost, vector, NULL);
    }
    else{
      /*ho letto un altro carattere lo metto a posto*/
      vector[k] = leggi;
      k++;
    }
    break;
  }
  default:
    assert(0);
  } /* switch */
 }while((j!=0));/*leggi fino all'EOF*/
 close(fd);
 return NULL;
}
void *scriviFile(char *nomeFile, double *tempo){
  int fd,j;
  fd = open(nomeFile,O_CREAT|O_WRONLY|O_APPEND, S_IRWXU);
  ERROR_1(fd,nomeFile);
  j =  write(fd,tempo,sizeof(double));
 ERROR_1(j,"fallita la write");
  close(fd);
  return NULL;
}
void *leggiFile(char *nomeFile){
```

```
  int fd,r;
  double tempo;
  fd = open(nomeFile,O_RDONLY);
  ERROR_1(fd,nomeFile);
  r = read(fd,&tempo,sizeof(double));
  while(r > 0){
    printf("--> %f\n",tempo);
    r = read(fd,&tempo,sizeof(double));
  }
  close(fd);
  return NULL;
}
void *calcolaMedia(char *nomeFile){
  int fd,r = 0,i;
  double tempo, contatore = 0,sommaParziale = 0,media = 0;
  char prova[LUNGHEZZAHOST];
  for(i=0;i<LUNGHEZZAHOST;i++) prova[i] = '\0';
  fd = open(nomeFile,O_RDONLY);
  ERROR_1(fd,nomeFile);
  r = read(fd,&tempo,sizeof(double));
  while(r > 0){
    contatore++;
    sommaParziale = sommaParziale + tempo;
    r = read(fd,&tempo,sizeof(double));
  }
  if(contatore!=0){
    media = sommaParziale/contatore;
    strcat(prova,nomeFile);
    strcat(prova,"Media");
    scriviFile(prova, &media);
  }
```

```
  close(fd);
  return NULL;
}
```

A.2.7 mainMaster.c

Il `mainMaster.c` contiene il main del Modulo Master. Non fa altro che effettuare controllo sui parametri passati dall'utente e esegue la `k = manager(host,task,atoi(argv[1]),argv[2]);` che è il nucleo di tutto il progetto. Di questa procedura ne parliano più avanti.

```
#include"manager.h"
int main(int argc, char *argv[])
{
  TC_lista *host = (TC_lista *)malloc(sizeof(TC_lista));
  TC_lista *task = (TC_lista *)malloc(sizeof(TC_lista));
  double k;
  int i;
  if(argc != 3){
perror("ESEGUIRE ./eseguibile numeroHostDaEseguire fileHost");
    exit(-1);
  }
  k = manager(host,task,atoi(argv[1]),argv[2]);
  if(k == -1){
    scriviFile("errore", &k);
    return 0;
  }else{
    scriviFile(argv[1], &k);
  }
  return 0;
}
```

A.2.8 manager.c

il file `manager.c` contiene tutte le procedure utilizzate nel Modulo Master in particolare le più importanti sono: la procedura `double manager(...)` che è il nucleo del modulo Master si occupa di effettuare tutta la fase di inizializzazione e inoltre si occupa di creare tutti i thread necessari (uno per ogni modulo Slave) che eseguono la procedura `worker(...)`. La funzione `worker(...)` si occupa di inviare il task allo Slave remoto. Questo viene fatto eseguendo le funzioni `creaClienteRemoto(nomeHost)` e `eseguiInRemoto(taskEstratto, c)`. Terminati tutti i thread viene calcolato il tempo di completamento del MOdulo Master e stampato al video.

```
#include "manager.h"
#include"remoto.h"
#include"timeutil.h"
#include "error.h"
#include"task.h"
#include <assert.h>
double k=0,ma=0;
/*SEMAFORI!*/
sem_t sem2,sem3,sem4,sem5;
void *inizializzaMutex()
{/*per i thread*/
  ERROR_1(sem_init(&sem2,0,0),
                        "inizializzando il semaforo POSIX");
  /*semaforo per settare i parametri da passare al thread*/
  ERROR_1(sem_init(&sem3,0,0),
                        "inizializzando il semaforo POSIX");
  ERROR_1(sem_post(&sem3),"V del semaforo");
  /*semaforo per settare i parametri dela lista TASK*/
  ERROR_1(sem_init(&sem4,0,0),
                        "inizializzando il semaforo POSIX");
  ERROR_1(sem_post(&sem4),"V del semaforo");
```

```
  /*semaforo per scrivere i risultati!*/
  ERROR_1(sem_init(&sem5,0,0),
                          "inizializzando il semaforo POSIX");
  ERROR_1(sem_post(&sem5),"V del semaforo");
return NULL;
}
struct uscita *inizializzaRisultati(int hostDaEseguire)
{
  struct uscita *tmp;
  tmp=(struct uscita *)
               malloc(sizeof(struct uscita)*hostDaEseguire);
  return tmp;
}
parametri *setParametri(TC_lista *listHost,
    TC_lista *listTask,struct uscita *ris, int threadIesimo)
{
  parametri *p = (parametri *)malloc(sizeof(parametri));
  lista *hostEstratto = (lista *)malloc(sizeof(lista));
  sem_wait(&sem3);
  /*debug  stampa(listHost->testa);*/
  if(listHost == NULL){
    ERROR_1(sem_post(&sem3),"V del semaforo");
    return NULL;
  }else{
    if(listHost->testa == NULL){
    ERROR_1(sem_post(&sem3),"V del semaforo");
    return NULL;
    }else{
      hostEstratto = get_Mutex(listHost);
   if(hostEstratto == NULL){
     ERROR_1(sem_post(&sem3),"V del semaforo");
```

```
      return NULL;
    }else{
      p->nomeHost=(char *)malloc(sizeof(char)*LUNGHEZZAHOST);
      strcpy(p->nomeHost,hostEstratto->nome);
      p->listaTask = listTask;
      p->risultatiClient = ris;
      p->threadCorrente = threadIesimo;
      free(hostEstratto);
      ERROR_1(sem_post(&sem3),"V del semaforo");
      return p;
    }
     }
   }
}
void * worker(void * par)
{
  CLIENT *c;
  parametri *my_par = ((parametri *)par);
  struct ingresso *taskEstratto =
         (struct ingresso *)malloc(sizeof(struct ingresso));
  lista * get = (lista *)malloc(sizeof(lista));
  char nomeHost[LUNGHEZZAHOST];
  strcpy(nomeHost,my_par->nomeHost);
  c = creaClienteRemoto(nomeHost);
  if(c == NULL){
    ERROR_1(sem_post(&sem2),"V del semaforo");
    return NULL;
  }else{
      get = get_Mutex(my_par->listaTask);
      if (get == NULL){
        taskEstratto = NULL;
```

```
}else{
  memcpy(taskEstratto->vectorIN,get->arrayInteri,
                          sizeof(double)*DIMENSIONE);
}
while(taskEstratto !=NULL){ /*finche' ci sono task*/
  my_par->risultatiClient =
                     eseguiInRemoto(taskEstratto, c);
  if(my_par->risultatiClient == NULL){
          ERROR_1(sem_post(&sem2),"V del semaforo");
            return NULL;
    }else{
  /*
    la struttura restituita e' questa:
    &my_par->risultatiClient
    per ora non fa nulla ogni volta che esguo un
    task (con eseguiInRemoto)la sovrascrivo quindi
    come risultato restituisco l'ultima
    PER ESEMPIO SCRIVERE I RISULTATI IN UN FILE
    DOVE SI POSSONO PRENDERE!
  */
  /*printf(" %i~",my_par->threadCorrente);*/
  /*stampaUscita(&my_par->risultatiClient);
   */
     }
  get = get_Mutex(my_par->listaTask);
  if (get == NULL){
    taskEstratto = NULL;
  }else{
  memcpy(taskEstratto->vectorIN,get->arrayInteri,
                          sizeof(double)*DIMENSIONE);
  /*debug
```

```
        stampa(get);
        stampaIngresso(taskEstratto);
        */
        }
      }
      ERROR_1(sem_post(&sem2),"V del semaforo");
    return (void *)nomeHost;
  }
}
double  manager(TC_lista *pool_Host,TC_lista *pool_Task,
                  int numeroHostDaEseguire, char *file)
{
  struct timeval *risultato =
        (struct timeval*)malloc(sizeof(struct timeval));
  lista *estratto;
  struct uscita *risultatiSlave;
  void **res;
  pthread_t *tid;
  double tempoDiCompletamento,error = -1;
  int h = 0,retcode;
  int j = 0;
  int t = 0; /*se ho t<h host devo fare t wait e non h
                      e cosi  devo aspettare t thread*/
  struct timeval start,stop;
  double k = 1000000;
  parametri *par ;
  int slaveFalliti=0;
  /* Alloca spazio per i puntatori ai risultati: */
  res = calloc(sizeof(void*), numeroHostDaEseguire);
  inizializzaSmaforiLista();
  inizializzaMutex();
```

```
risultatiSlave =
           inizializzaRisultati(numeroHostDaEseguire);
creaTask(pool_Task);
memorizzaFile(pool_Host,file);

tid = (pthread_t *)
   malloc(sizeof(pthread_t)*(numeroHostDaEseguire+1));
par = (parametri *)malloc(sizeof(parametri)+1);
gettimeofday(&start,NULL); /* la prima e' per la cache*/
gettimeofday(&start,NULL);
while(h < numeroHostDaEseguire){
  par = setParametri(pool_Host,pool_Task,risultatiSlave,h);
  /*printf(" PRIMA DELLA CREAZIONE DEL THREAD ============>
                                      %s\n",par->nomeHost);*/
  if(par != NULL){
    /*creo i numeroHostDaEseguire thread*/
retcode = pthread_create(&tid[h], NULL, worker, (void *)par);
    ERROR_1(retcode,"errore creando il THREAD\n");
    h++;t++;
  }else{/*sono finiti gli host!*/
   h = numeroHostDaEseguire;
  }
}
for(j=0;j<t;j++){
  /*printf("sono wait t=%i j=%i\n",t,j)*/;
  sem_wait(&sem2);
}
/*  printf(" aspetto che i  thread teminino \n");*/
/*aspetto che i thread siano terminati*/
for(j=0;j<t;j++){
  retcode = pthread_join(tid[j], &(res[j]));
```

```
    if(res[j]==NULL) {
      slaveFalliti++;
      /* printf(" il thread %i  host %s==>  NULL\n"
                                  ,j,(char *)res[j]);*/
    }else{
      /*printf("Il thread %i ha restituito l'oggetto in
      %p, vale  %s\n", j, (void*)res[j], (char *)res[j]);*/
    }
  }
  gettimeofday(&stop,NULL);
  timeval_subtract (risultato, &stop, &start);
  tempoDiCompletamento=
             (((risultato->tv_sec*k)+risultato->tv_usec)/k);
  /*printf("\ntempoDiCompletamento=%f\n",
                                    tempoDiCompletamento);*/
  estratto = estrai(pool_Host);
  while( estratto != NULL ){estratto = estrai(pool_Host);}
  free(risultato);
  free(par);
  printf(" slaveFalliti = %i t = %i h = %i tempo = %f",
                      slaveFalliti,t,h,tempoDiCompletamento);
    printf("\nDIMENSIONE=%iNITER=%i<---\n",DIMENSIONE,NITER);
  if(t<h)return error;
  if(slaveFalliti!=0){
    /*almeno uno slave e' fallito i risultati sono sballati!*/
    return error;
  }else{
    return tempoDiCompletamento;
  }
}
```

A.2.9 remoto.c

il file `remoto.c` contiene le funzioni che eseguono le chiamate remote al Modulo Slave in particolare troviamo `CLIENT *creaClienteRemoto(char *host)` che crea la connessione con l'host attraverso la `cl = clnt_create(host, ARRAY_PROG, ARRAY_VERSION, tcp);` inoltre verifica che non ci siano stati fallimenti attraverso la `clientNull(CLIENT *cliente, char *host)`. La `struct uscita *eseguiInRemoto(struct ingresso *in, CLIENT *cl)` la quale esegue la `consumatask_1(in,cl);` che si connette direttamente alla funzione `consumatask_1_svc(...);` in esecuzione sul Modulo Slave.

```
#include"remoto.h"
CLIENT * clientNull(CLIENT *cliente, char *host)
{
  if(cliente == NULL){
    clnt\_pcreateerror(host);
    return NULL;
  }else{return cliente;}
}
CLIENT *creaClienteRemoto(char *host)
{
  CLIENT *cl;
  /*printf("CREA CLIENT REMOTO = %s\n",host);*/
  cl = clnt\_create(host, ARRAY\_PROG, ARRAY\_VERSION, "tcp");
  cl = clientNull(cl,host);
  return cl;
}
struct uscita
         *eseguiInRemoto(struct ingresso *in, CLIENT *cl){
  return (struct uscita *)consumatask\_1(in,cl);
}
```

A.2.10 slave.c

il file `slave.c` contiene le funzioni necessarie all'esecuzione del Modulo Slave. Attende che il Modulo Master gli invii il task da eseguire, lo esegue e restituisce il risultato. In questo caso particolare lo slave esegue la funzione `usc = calcolTask(ingr)`

```
#include "timeutil.h"
#include "file.h"
#include "array.h"
#include"task.h"

double t = 0;
struct uscita *consumatask_1_svc(struct ingresso *ingr,
                                        struct svc_req *req)
{
  struct uscita *usc =
            (struct uscita *)malloc(sizeof(struct uscita));

  usc = calcolaTask(ingr);

return usc;
}
```

A.2.11 task.c

il file `task.c` esegue tutte le funzioni relative ai task. Nel file troviamo la `void *creaTask(TC_lista *list)` che genera i task da eseguire, la `struct uscita *calcolaTask(struct ingresso *ingr)` che esegue i task e inoltre troviamo altre funzioni che servono a stampare le `struct ingresso, struct uscita` e una funzione `char * DaInteriAChar(int intero)` che trasforma gli interi in char

```
#include "lista.h"
#include <math.h>

int successivo=0;

char * DaInteriAChar(int intero)
{
  double d;
  int contatore,l,n,i;
  char *k;

  i = intero;
  d = (double) intero;
  contatore = 0;
  while(i >0)
    {
      i = i/10;
      contatore++;
    }
  k = ecvt(d,contatore,&l,&n);
  return k;
 }

void *creaTask(TC_lista *list)
```

```
{
  int i,j=0;
  double array[DIMENSIONE];
  char *numero;
  char nomeTask[LUNGHEZZAHOST];

  for(i=0;i<NTASK;i++){
    for(j=0;j<DIMENSIONE;j++){
      array[j] = successivo;
      successivo++;
    }/*cosi'i task sono tutti !=*/
    numero = DaInteriAChar(i);
    strcpy(nomeTask,"task");
    strcat(nomeTask,numero);
    inserisci(list,nomeTask,array);
    }
  /*debug
  printf("stampiamoli per vedere se sono giusti\n");
  stampa(list->testa);
  printf("google\n");
  exit(0);
  */
  return NULL;
    }


struct uscita *calcolaTask(struct ingresso *ingr)
{
  int i,j;
  struct uscita *usc =
              (struct uscita *)malloc(sizeof(struct uscita));
```

```
  for(i=0;i<DIMENSIONE;i++){
    usc->vectorOUT[i] = sin(ingr->vectorIN[i]);
    for(j=0;j<(NITER/DIMENSIONE);j++){
      usc->vectorOUT[j%DIMENSIONE] =
                          sin(usc->vectorOUT[j%DIMENSIONE]);
      }
    }
    return usc;
  }
}
void *stampaIngresso(struct ingresso *ingr)
{
  int i;
  printf("========== %i ====================\n",ingr->Niter);
  for(i=0;i<DIMENSIONE;i++) printf("| %f",ingr->vectorIN[i]);
  printf("\nI======= FINE ================\n");
  return NULL;

}
void *stampaUscita(struct uscita *usc)
{
  int i;
  printf("U============================================\n");
  for(i=0;i<DIMENSIONE;i++)printf("|  %f",usc->vectorOUT[i]);
  printf("\nU============== FINE ======================\n");
  return NULL;
}
```