dPOV-Ray
Distributed Persistence-of-Vision RayTracer

Relazione di Aldo Romani, 2146 55863
mat. 2146 55863, email: aldrom@tin.it

Introduzione

Il ray tracing è un metodo che consente di realizzare al calcolatore immagini fortemente realistiche.

> *"Ray-tracing is a rendering technique that calculates an image of a scene by simulating the way rays of light travel in the real world. However it does its job backwards. In the real world, rays of light are emitted from a light source and illuminate objects. The light reflects off of the objects or passes through transparent objects. This reflected light hits our eyes or perhaps a camera lens."* (dalla documentazione POV-Ray)

Si parte da una descrizione della scena che vogliamo rappresentare, realizzabile di solito mediante pacchetti di modellazione interattiva oppure mediante la stesura manuale di file di testo redatti secondo la sintassi di un linguaggio di modellazione. In entrambi i metodi sarà necessario specificare quali oggetti compongono la nostra scena, quale forma hanno, dove si trova-

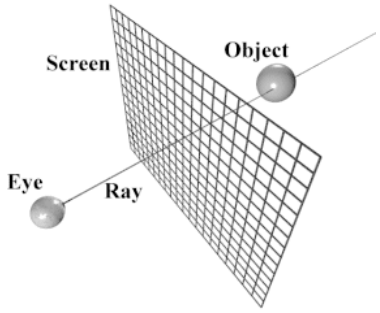


Fig.1 Rappresentazione della scena

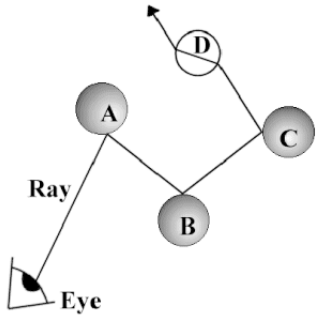


Fig.2 Tracciamento dei raggi

no, di che colore sono, con quale texture sono ricoperti e, importante, dove sono collocate e di che tipo sono le sorgenti luminose che li illuminano. A questo punto, noti anche il punto e l'angolo di visuale, l'applicativo di ray-tracing è in grado di calcolare l'aspetto della scena, intesa come una matrice di pixel, dandone una adeguata resa visiva.

L'algoritmo di ray-tracing è piuttosto complesso ed oneroso computazionalmente: modella matematicamente i raggi di luce nel loro "viaggio" attraverso la scena. Per ogni punto della matrice di visualizzazione viene "lanciato" dal punto di visualizzazione un raggio di luce, e ne vengono calcolate le intersezioni con gli oggetti della scena. L'intersezione più vicina determinerà il colore del punto considerato. Inoltre per questa sarà necessario calcolare quante sorgenti luminose sono direttamente visibili per stabilire se il punto si trova o meno in una zona d'ombra. Inoltre per ogni intersezione si dovrà anche tenere conto di fenomeni di riflessione e di rifrazione. Questo algoritmo implica un numero molto elevato di operazioni floating-point

Per un approfondimento sul ray-tracing e sui dettagli dell'algoritmo è possibile consultare *"The Recursive Ray Tracing Algorithm"* di Jamis Buck, disponibile su web e allegato a questa relazione.

Appartiene alla categoria dei ray-tracers POV-Ray (Persistence of Vision Raytracer), applicativo copyrighted freeware disponibilie in rete insieme al suo codice sorgente (http://www.povray.org)

POV-Ray è in grado di effettuare il rendering mediante raytracing di scene descritte mediante files di testo nel suo linguaggio proprietario (POV-Ray Scene Language), piuttosto semplice ed intuitivo. L'elaborazione prevede una fase iniziale di parsing ed una fase di rendering.

Il motore di rendering di POV-Ray sfrutta pienamente solo un processore di sistema, la stessa documentazione suggerisce per sistemi SMP di mettere in esecuzione diverse copie del programma, ognuna su una diversa CPU, e ad ognuna fare eseguire una porzione del rendering.

Il progetto realizzato consiste in una versione distribuita del ray-tracer, in grado di supplire a questo tipo di carenze, viste però nell'ottica di un sistema distribuito costituito da una rete di calcolatori .

In seguito ad un'analisi del codice sorgente di POV-Ray è stata isolata la routine principale, utilizzata nel progetto come motore di rendering, senza entrare nei dettagli della realizzazione dell'algoritmo.

L'idea principale è quella di realizzare una struttura che implementi un modello di tipo farm, con un servitore che coordina il lavoro di una serie di slaves, e fornisce un'interfaccia per l'uso del servizio (vedi Figura 3). In questo modo sarà possibile sfruttare la potenza di calcolo di una molteplicità di processori, riducendo i tempi (lunghi) necessari al rendering. Il coordinatore dovrà alla fine ricostruire l'immagine unendo i segmenti forniti dagli slave.

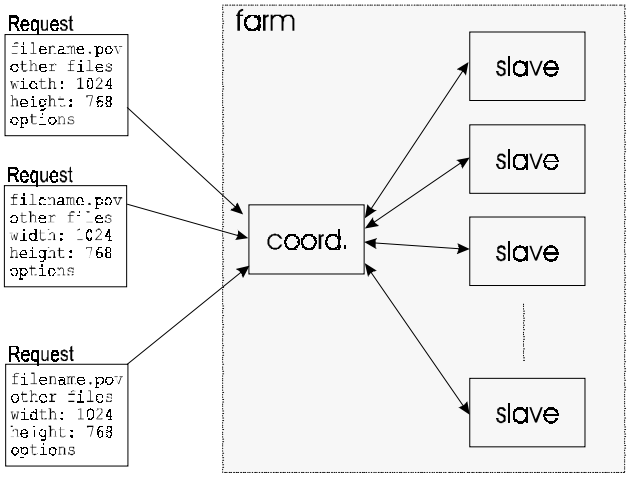


Fig.3 Schema di principio di dPOV

Realizzazione e caratteristiche del progetto

L'idea principale che caratterizza il progetto è l'utilizzo di una molteplicità processori, disponibili sui vari nodi di una rete, per partecipare al rendering di una scena mediante ray-tracing. L'algoritmo di ray-tracing è infatti fortemente CPU-bound e non può che giovarsi della presenza contemporanea di diversi nuclei di elaborazione indipendenti.

Sarà necessaria allora la presenza di un processo *Coordinatore*, che sappia attuare opportune politiche di *Load Sharing* e *Load Balancing*, al fine di ottimizzare l'impiego di CPU, distribuendo in maniera appropriata il lavoro tra gli slaves e sapendo reagire agli eventi del caso (arrivo di alcuni nuovi slaves liberi, fallimento o completamento di altri) ed eventualmente tentando di fare, se questo converrà, una riallocazione dinamica del carico. Queste politiche sono state realizzate mediante allocazione, disallocazione, suddivisione e trasferimento di *job* tra i nodi *Slave* disponibili (brevemente, con job si intende in questo progetto una porzione di un'immagine da renderizzare, il concetto verrà analizzato in dettaglio successivamente).

Come già detto in precedenza, la realizzazione del progetto si basa sull'estrazione dal codice sorgente di POV-Ray della routine principale di rendering, e sul suo inserimento in un processo di tipo *Slave*. Brevemente, un processo *Slave* si rende disponibile ad eseguire del rendering, e comunica questa sua disponibilità ad un *Coordinatore.* Lo Slave resterà poi in attesa che gli venga assegnato un *job* dal coordinatore. Uno *Slave* non può eseguire più di un job contemporaneamente: l'obiettivo è infatti l'ottimizzazione dello sfruttamento delle CPU, e visto che anche un rendering dei più semplici sfrutta quasi il 100% del tempo di CPU disponibile si può notare che non vi saranno significativi vantaggi allocando più job su un singolo nodo.

Il progetto è stato realizzato in linguaggio C, su piattaforma Win32, senza fare uso di particolari framework o classi (es. MFC, etc.), ed è un'applicazione mista Console/GUI. E' stato collaudato e "debuggato" su una LAN di 3 PC Win9x (con CPU Intel P200MMX, P133, DX4-120). Sono state utilizzate le funzionalità della Standard C Library, dell'API Win32, e delle Windows Sockets, dunque è stato necessario costruire l'applicazione a partire dagli elementi più elementari. La Standard C Library è stata usata in particolare per il Console I/O, e l'allocazione della memoria. Dell'API Win32 sono state sfruttate le funzionalità multi-threaded, i *synchronizaztion objects*, e la gestione base di finestre e bitmap. Tutto lo strato di comunicazioni fra nodi è stato implementato mediante le Windows Sockets (fortemente aderenti allo standard delle Berkerly Sockets).

Prima di continuare qualche rapido cenno alla struttura di comunicazione tra *Coordinatore* e *Slaves*
Coordinatore e *Slaves* possono scambiarsi diversi tipi di messaggi:
1) messaggi di servizio
2) messaggi del protocollo di rendering
3) flussi di bytes

I messaggi dei tipi 1) e 2) vengono scambiati attraverso il protocollo UDP, ed hanno il seguente formato:

```
typedef struct packet
{
     unsigned long int datagram_id;
     unsigned long int command;
     char data[312];
} PACKET;
```

I messaggi di servizio sono caratterizzati da un datagram_id pari a 0, mentre per i messaggi del protocollo di rendering viene usata una numerazione progressiva per supportare tolleranza ai guasti e perdita di messaggi, e per gestire eventuali ritrasmissioni. Appartengono alla categoria dei messaggi di servizio i messaggi di resa disponibilità o di fallimento di uno slave, i broadcast del master, i messaggi del tipo "Are you alive" e le relative risposte. Questi messaggi vengono ricevuti e

gestiti indipendentemente dalla fase di avanzamento del protocollo di rendering. I messaggi del protocollo di rendering sono invece marcati con una numerazione progressiva del datagram_id. *Coordinatore* e *Slave* sono in ascolto rispettivamente sulle porte UDP contrassegnate come COORD_UDP_PORT e SLAVE_UDP_PORT, stabilite negli header files dell'applicazione.

Dovranno anche essere trasmessi flussi di bytes, ad esempio i files necessari per il rendering dovranno essere trasferiti presso lo slave, e questo dovrà trasferire la bitmap man mano che viene elaborata. Per questo tipo di flussi è stato stabilto di usare uno streaming affidabile TCP. E' lo *Slave* a restare in attesa di connessioni da parte del coordinatore.

Passiamo ora ad analizzare più in dettaglio i componenti principali del progetto:

- la struttura del *Coordinatore*
- la struttura degli *Slaves*
- il protocollo di comunicazione e interazione
- la tolleranza ai guasti

Struttura del Coordinatore

Il *Coordinatore* è un'applicazione mista console/GUI, utilizza la console per visualizzare messaggi relativi alla comunicazione, al bilanciamento del carico, allo stato dei nodi slaves. Le finestre principali sono mostrate in Figura 4.

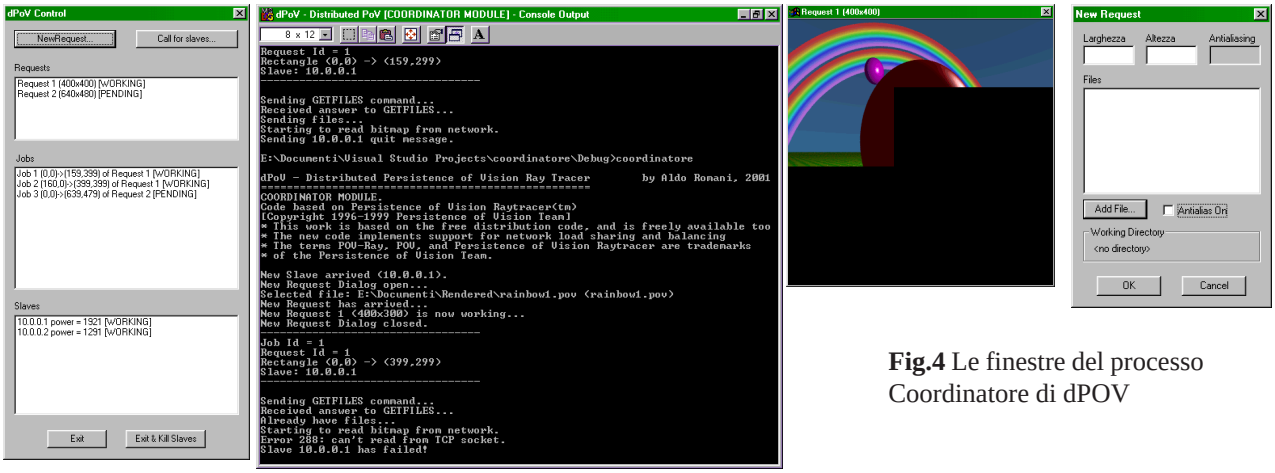


Fig.4 Le finestre del processo Coordinatore di dPOV

Il *Coordinatore* può accettare nuove richieste di rendering, e resta in attesa che qualche slave comunichi la propria disponibilità. Richieste pervenute in assenza di slave disponibili saranno marcate come pendenti (PENDING), le altre saranno etichettate come in fase di svolgimento (WORKING). Un coordinatore può stimolare la comunicazione di disponibilità degli slave inviando un broadcast sulla rete locale. Tutti gli slaves disponibili risponderanno comunicando la propria disponibilità.

Per tenere traccia delle richieste è stato implementato il tipo di dati REQUEST. Una richiesta è caratterizzata da un identificatore univoco (request_id), dall'elenco dei files necessari al rendering, dalle opzioni necessarie al rendering (dimensione delle bitmap da creare, livello di antialiasing, etc.), dal suo stato (PENDING, WORKING) e da altri campi necessari internamente all'applicativo. Viene mantenuta una Tabella delle Richieste

```
typedef struct request
{   unsigned long int request_id;
    int status;
    unsigned int size_x, size_y;
    char AA[5];
    int AAon;
    ...        } REQUEST;
```

Il *Coordinatore* è inizialmente composto da due thead:

- un thread Scheduler, che gestisce il Load Sharing and Balancing
- un thread Receiver, che resta in ascolto sulla porta UDP assegnata al servizio
- inoltre per ogni job attivato viene creato un ulteriore thread di tipo DoJob che si occupa delle fasi del protocollo di rendering

Il thread Receiver resta in attesa di un qualunque messaggio sulla porta UDP, discrimina tra messaggi di servizio e messaggi di protocollo. Se arriva un messaggio di servizio, reagisce di conseguenza, ad esempio, arriva un messaggio di tipo "Are you alive?", risponde con "I am alive", oppure se si aggiunge un nodo slave notificherà l'evento al thread Scheduler. I messaggi di protocollo verranno invece smistati ai thread di competenza.

Il thread Scheduler resta sospeso in attesa finchè non si verifica uno di questi eventi:
- viene aggiunta una nuova richiesta di rendering
- un nuovo slave comunica la propria disponibilità
- uno slave e il relativo job sono falliti
- uno slave ha completato il job assegnatogli

Il primo tipo di eventi consegue ad un input dell'utente, mentre gli altri vengono generati dal thread di ricezione che ha ricevuto dalla rete messaggi corrispondenti. Eventi di fallimento di job possono anche essere generati dai thread *DoJob* in caso di timeout di trasmissione/ricezione ed errori di protocollo

Funzionamento dello scheduler

Vediamo ora in dettaglio il funzionamento dello Scheduler, che è il cuore del sistema di Load Sharing and Balancing.

Lo scheduler deve, a partire dalle richieste in arrivo e da quelle correntemente in esecuzione, noti lo stato di disponibilità degli slaves e la stima della loro potenza di calcolo, studiare un piano di allocazione sui vari slaves.

Ogni richiesta deve essere suddivisa in uno o più jobs (porzioni rettangolari di immagini) di dimensioni opportune compatibili con la potenza di calcolo ed il numero degli slaves disponibili. Un esempio è mostrato in figura.

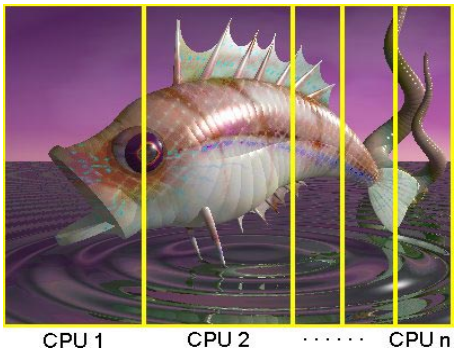


Fig.5 Suddivisione dell'immagine tra gli slaves

Un *job* è definito nel modo seguente:

```
typedef struct job
{     unsigned long int job_id;
      unsigned long int request_id;
      HANDLE ServingThread;
      RECTANGLE rect;
      int status;
      int secsleft;
      SLAVE sl;
      PACKET LastPacketSent
      PACKET LastPacketReceived;
      HANDLE NewPacketEvent;
} JOB;
```

Gli slaves disponibili sono mantenuti in una Tabella degli Slaves. Lo stesso viene fatto per i job: avremo una Tabella dei Jobs. Insieme alla Tabella delle Richieste consentono allo scheduler di operare. Le chiavi primarie di ricerca in queste tabelle saranno i rispettivi job_id, request_id, e l'indirizzo IP dello slave. Ogni entry di ciascuna tabella potrà, tramite operazioni di join essere associata ad una entry delle altre due tabelle.

Algoritmo di Allocazione.

E' stato affrontato il problema generico di allocare il rendering di N rettangoli di dimensioni date su M processori di potenza data, con $N \leq M$

Si è valutato anche se vale la pena utilizzare tutti i processori, o meno. Infatti allocare un job su un nodo slave comporta un certo overhead, dato dal tempo di trasmissione dei files, dall'overhead di parsing remoto dei files, e dalla ricezione della bitmap dalla rete. Questo overhead potrebbe superare la differenza di tempo necessario al rendering rispetto al caso di assenza dello slave. Potrebbe dunque non convenire sfruttare tutti i nodi disponibili.

L'Algoritmo di Allocazione risolve anche questo problema, oltre a trovare la dimensione ottimale del rendering, sfrutta

	R_1	R_2	R_3	…	R_N
P_1	x_{11}	x_{12}	x_{13}	…	x_{1N}
P_2	x_{21}	x_{22}	x_{23}	…	x_{2N}
.	.	.	.		.
.	.	.	.		.
P_M	x_{M1}	x_{M2}	x_{M3}	…	x_{MN}

$$0 \leq x_{ij} \leq 1$$

tecniche di ottimizzazione lineare

Supponiamo di avere N rettangoli $R_1, R_2, ..., R_N$ ognuno dei quali ha dimensioni (w_i, r_i)

Supponiamo inoltre di avere M processori $P_1, P_2, ..., P_M$ di cui sia nota la potenza p_j in termini di pixel/sec.

Il problema dell'allocazione ottima consiste nel trovare i valori alle incognite x_{ij} mostrate in tabella che minimizzazano il tempo necessario e contemporaneamente soddisfano alcuni vincoli. L'incognita x_{ij} indica quale frazione del rettangolo i deve essere assegnata al processore j.

Le incognite dovranno sottostare una serie di condizioni:

1) condizioni "orizzontali": un processore può eseguire soltanto un job

$$\forall j, 1 \leq j \leq M: \quad (x_{j1}, x_{j2}, \quad , x_{jN}) = (0,0, \quad ,0,1,0, \quad ,0)$$

2) condizioni “verticali”: tutti i rettangoli assegnati dovranno essere renderizzati, per cui:

$$\forall i, 1 \le i \le N: \quad \sum_{j=1}^{M} x_{ij} = 1$$

Il programma calcolerà per ogni possibile combinazione degli x_{ij} gli overhead dei vari processori OH(1), OH(2), ... , OH(M), valutati secondo la formula :

$$OH(i) = OH_0 + \alpha \cdot sizeof(FILES) + \beta \cdot sizeof(RENDERING)$$

I tempi di esecuzione stimati dei vari processori saranno dati dalla seguente espressione

$$T_j = \frac{1}{p_j} \cdot \sum_{i=1}^{N} \left[x_{ji} \cdot area(R_i) \right] + OH(j)$$

Il tempo complessivo stimato del rendering sarà quello dello slave più lento: $T = \max\{T_1, T_2, \quad , T_M\}$

Il calcolo viene ripetuto per ogni combinazione delle N•M incognite x_{ij}. La configurazione ottima sarà quella corrispondente al minimo tempo calcolato: $T_{opt} = \min\{T\}$

La complessità dell’algoritmo è esponenziale, in quanto devono essere provate tutte le possibili combinazioni degli x_{ij}.

Questa complessità viene comunque ridotta sfruttando le condizioni 1) e 2).

Le condizioni 1) eliminano a priori moltissimi valori. Se l’intervallo [0,1] di variabilità di x_{ij} viene discretizzato in Q valori si passa dalle Q^N combinazioni per linea alle Q•N combinazioni imposte dal vincolo considerato.

Le condizioni 2) riducono le linee da considerare da M a M-1, in quanto l’ultima linea è automaticamente determinata, visto che il risultato della somma di ogni colonna deve dare 1,

Sperimentalmente l’algoritmo diventa computazionalmente oneroso grosso modo quando N+M>6, comunque si vedrà che questa combinazione sarà piuttosto improbabile durante l’utilizzo del software. Le combinazioni più probabili saranno del tipo (N=1, M>1) e (M=1,N>1). Si veda a questo proposito il paragrafo dedicato allo Scheduler.

L’algoritmo ci consente di trovare quali job da allocare sugli slaves. Ovviamente, se ho più richieste di quanti slaves sono dsponibili e liberi, alcune di queste resteranno in sospeso. Parimenti, sarà possibile che uno slave non si veda allocato nessun carico, poichè questa situazione sarà stato giudicata più conveniente.

Gestione degli eventi da parte dello scheduler

Lo *Scheduler* resta in attesa di uno di questi eventi:

1- viene aggiunta una nuova richiesta di rendering
2- un nuovo slave comunica la propria disponibilità
3- uno slave e il relativo job sono falliti
4- uno slave ha completato il job assegnatogli

Vediamo i vari casi in dettaglio:

1. Se si aggiunge una nuova richiesta:
- se non ci sono slave inattivi allora segno la richiesta come in atto, creo un job di dimensione pari alla dimensione dell’intera immagine, marco il job come pendente ed esco
- se ci sono slaves inattivi alloco la richiesta su di essi generando i jobs necessari per l’ottimizzazione

2. Se un nuovo slave comunica la propria disponibilità
- se non ci sono job pendenti e in fase di esecuzione aggiungo lo slave come libero alla Tabella degli Slaves
- se ci sono job pendenti ne assegno uno
- se ci sono dei job in fase di esecuzione, si cerca il job che ha il tempo di completameno più lungo in base alla velocità attuale di rendering. Se si può avere un significativo vantaggio interrompo questo slave e suddivido quello che rimane dell’immagine tra il nuovo e il vecchio slave. Se non c’è vantaggio aggiungo lo slave alla lista dei job liberi (questo caso è mostrato in Figura 6. Si noti come un terzo slave interrompa il job più lento e partecipi con lui al rendering della frazione rimanente

Il vantaggio è valutato in questo modo: si calcola il tempo stimato rimanente del job più lento, se questo è significativamente distante dalla media dei tempi di completamento di tutti i job, e se è anche maggiore di una soglia di tempo fissata (es.20 sec) allora ho vantaggio in caso contrario no.

3. Se uno slave e il relativo job sono falliti allora cancello lo slave dalla relativa tabella, e segno il job come fallito (dicitura circa equivalente a pendente). Se ho degli slaves liberi alloco il job fallito su questi, marcandolo come in fase di esecuzione In figura 7 è mostrato il caso di fallimento di un job.

4. Se uno slave ha completato il job assegnatogli il comportamento è simile a quello del punto 1. Se converrà interrompere un job, questo verrà fatto e il rettangolo rimanente suddiviso tra i due, altrimenti verrà marcato come IDLE.

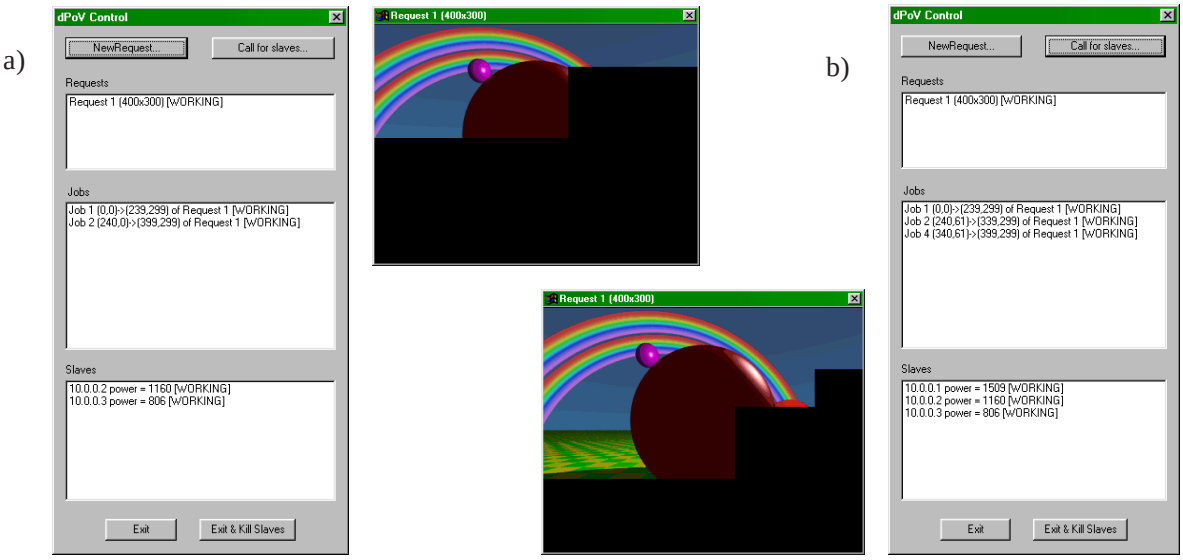


Fig.6 Riallocazione del carico. **a)** due slaves stanno lavorando **b)** un terzo slave si rende disponibile, uno degli slave attivi viene interrotto, e il rettangolo rimanente allocato tra questo e il nuovo arrivato

Per ogni job che viene assegnato ad uno slave, il coordinatore mette in esecuzione un thread supplementare DoJob che ha il compito di gestire il protocollo di rendering remoto. In modo particolare dovrà segnalare allo slave l'inizio del protocollo, spedire i files, se necessario, trasferire la bitmap, e chiudere il protocollo segnalando il completamento della richiesta. Maggiori dettagli saranno presenti nella sezione dedicata al protocollo di comunicazione

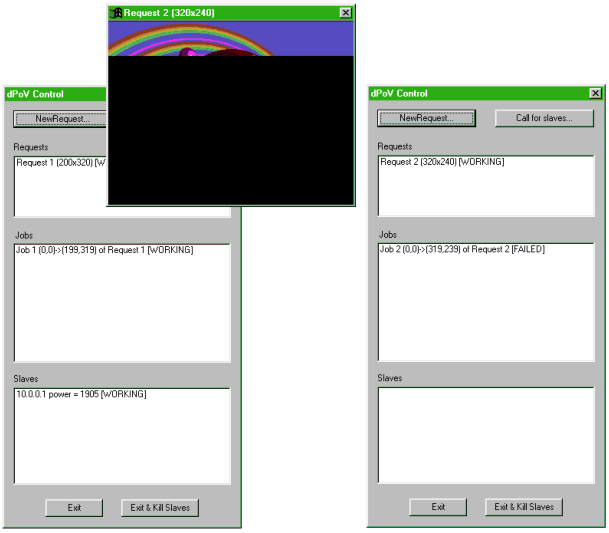


Fig.7 Fallimento di uno slave in assenza di sostituti

Valutazione della possibilità di effettuare Riallocazione Dinamica

E' stata valutata la possibilità di effettuare una *riallocazione dinamica* del carico, pensata in questi termini: lo scheduler in attesa di uno degli eventi descritti sopra fissa inizialmente un timeout.

Se non arrivano eventi e scade il timeout un nuovo thread esegue in background l'algoritmo di riallocazione del carico prendendo tutti i job pendenti e in esecuzione e allocandoli sugli slaves. Questo processo potrebbe essere lungo, vista la complessità esponenziale dell'algoritmo di allocazione, se ho un elevato numero di jobs e Slaves, dunque viene lasciato eseguire in background senza interferire con il normale funzionamento. Lo scheduler si rimette in attesa. Se uno dei 4 eventi previsti interrompe l'attesa prima di un nuovo timeout il thread viene cancellato, e si gestisce l'evento. Se il thread termina senza che nessun evento sia sopravvenuto, allora lo scheduler può mettere in atto la riallocazione, tenendo conto di alcune modifiche.

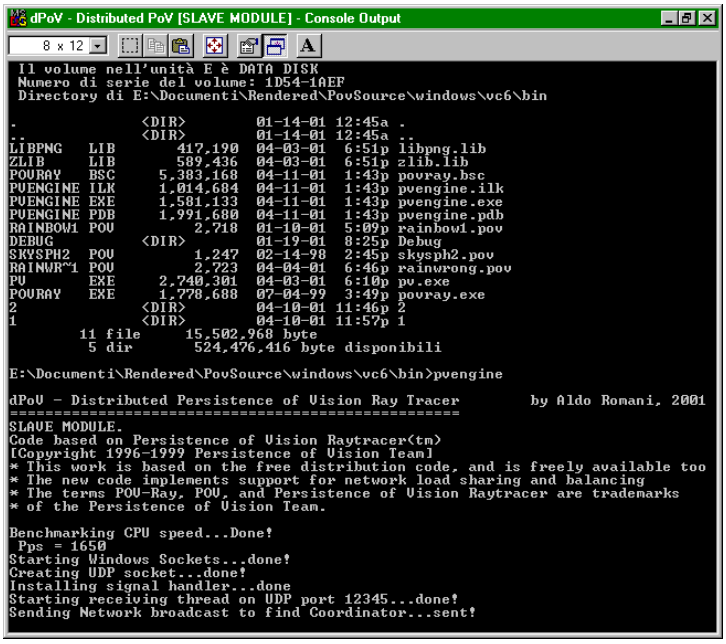
Infatti un job potrebbe essere terminato mentre il thread in background calcolava la riallocazione. Semplicemente non devo riallocare i rettangoli di immagine ottenuti dai corrispondenti job terminati, lasciando qualche slave libero. Inoltre sicuramente i job rimasti in esecuzione avranno ottenuto un ulteriore frazione della bitmap, per cui i rettangoli ottenuti dalla routine riallocazione dovranno essere scalati in proporzione.

Tuttavia questo tipo di riallocazione apporta solamente vantaggi marginali e complica la gestione degli eventi. Infatti le 4 condizioni previste dallo scheduler già sono sufficienti a garantire la piena occupazione dei processori disponibili e uno sfruttamento ottimale delle risorse di calcolo remote. L'unico vantaggio visibile che potrebbe essere apportato da questa variante è quello di evitare alcuni passi di successiva riallocazione conseguenti a terminazioni di job. Il vantaggio è dunque marginale vista l'esiguità dei tempi necessari allo scheduler per gestire le terminazioni dei jobs e per far partire delle nuove richieste. Dunque questo tipo di riallocazione dinamica non è stato incluso nel progetto

Thread di ricezione

Analizziamo il comportamento del thread di ricezione. Il thread di ricezione si mette in ascolto sulla porta COORD_UDP_PORT. Non appena arriva un messaggio su questa porta lo serve, rispondendo direttamente o segnalando l'evento al thread interessato (Scheduler o DoJob).

Tra i più importanti messaggi che vengono serviti direttamente possiamo trovare i messaggi di fallimento di uno slave. Se uno slave dichiara il fallimento, prima di segnalarlo allo scheduler dovrò aggiornare le Tabelle dei Job e degli Slaves



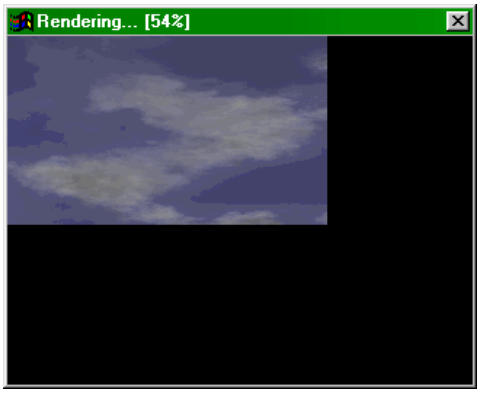


Fig. Schermata principale della Console dello slave,e finestra di rendering. Si noti che il job assegnato è solo una parte dell'immagine completa

Struttura dello Slave

Lo slave, abbiamo visto, contiene la routine principale del programma POV-Ray, implementata come thread indipendente che viene attivato by-need quando arriva una richiesta di rendering. Abbiamo poi altri due thread principali, sempre attivi:

- il thread principale
- il thread di ricezione

Il thread di ricezione è strutturato in maniera simile a quello del coordinatore: serve direttamente i messaggi di servizio (con datagram_id = 0) e smista i messaggi del protocollo di rendering al thread principale.

La porta UDP sulla quale resta in ascolto il thread di ricezione è la SLAVE_UDP_PORT, diversa da quella del coordinatore. I messaggi gestiti direttamente dal thread di ricezione sono i broadcast dei coordinatori in cerca di slaves liberi, i messaggi "Are you alive?", ed i messaggi di terminazione o fallimento del coordinatore. Se quest'ultima possibilità avviene lo slave subisce un reset e si riporta nella condizione iniziale di attesa di un nuovo coordinatore.

Il thread principale aspetta l'arrivo di un nuovo coordinatore (evento che gli viene segnalato dal thread di ricezione). La fase successiva è quella della gestione del protocollo di rendering (in cui il suo pari è il thread DoJob del coordinatore).

Lo Slave, cominciato il protocollo di rendering, inizializza una socket stream TCP, legandola alla porta SLAVE_TCP_PORT. A questa porta si collegherà il Coordinatore per trasferire i files necessari e la bitmap generata, e servirà anche per stabilire dei punti di sincronizzazione tra i processi attivi sui due nodi.

Una caratteristica importante dello slave è quella di cercare di effettuare una chiusura graceful, eliminando tutti i files temporanei prodotti, chiudendo tutte le connessioni di rete.e segnalando sempre al master un suo malfunzionamento o fallimento in modo che questo possa reagire prontamente. A tal fine è stata necessaria una gestione del segnale SIGINT conseguente a chiusura forzata dell'applicazione, implementato in Win32 in maniera simile all'ambiente Unix.

Stima della potenza di calcolo dello slave

Lo Slave, non appena comincia la sua esecuzione, tenta di fare una stima del tempo di CPU disponibile in quel momento, che comunicherà al Coordinatore nella prima fase del protocollo di rendering. Viene effettuato un mix di operazioni floating-point a partire da numeri casuali su un array di flot insieme ad una serie di operazioni ricorsive (calcolo delle sequenze di Fibonacci di alcuni numeri). Questo tipo di operazioni (floating-point e chiamate ricorsive) cercano di simulare quello che avviene all'interno del motore di rendering. Le operazioni floating-point servono a stimare la potenza di calcolo pura del microprocessore, mentre le chiamata ricorsive cercano di emulare l'overhead dovuto alle ripetute function-calls nei cicli iterativi/ricorsivi del renderer.

Per avere un stima del tempo impiegato vengono utilizzati i Performance Counter implementati dalla piattaforma Win32. Normalizzando il risultato rispetto al tempo impiegato per calcolare un'immagine fissata in condizioni nominali, cablato nel programma, si ottiene una stima dei pixel/sec. Non è necessario che la stima sia accurata, poichè diverse regioni dell'immagine possono avere complessità molto diverse tra loro, si pensi ad un cielo nitido, oppure ad un dettaglio di un tavolo su cui sono presenti lampade e oggetti vari. Dunque una stima approssimata della potenza di calcolo è sufficiente. Sarà il Coordinatore che dovrà riequilibrare la situazione mediante allocazioni/disallocazioni.

Protocollo di comunicazione

Analizziamo in dettaglio il protocollo di comunicazione tra Coordinatore e Slaves, insieme al formato e alla semantica dei messaggi che questi si scambiano.

L'unità fondamentale di cominicazione è il pacchetto (PACKET), che viene incapsulato in un datagramma IP dallo strato di comunicazione offerto dalle Windows Sockets, definito come segue:

```
typedef struct packet
{   unsigned long int datagram_id;
    unsigned long int command;
    char data[DATA_SIZE];   // DATA_SIZE non dovrebbe superare la dimensione
} PACKET;                   // massima di un datagramma IP per evitare frammentazione
```

Abbiamo visto che è possibile distinguere tra pacchetti di servizio, che hanno `datagram_id` pari a 0 e pacchetti del protocollo di rendering.

Il campo `command` permette di identificare l'azione che il destinatario è tenuto a fare. I vari tipi di comandi possibili sono così definiti nell'header files dell'applicazione.

```
#define ADDNODE_COMMAND 1
#define CALLFORSLAVES_COMMAND 2
#define GETFILES_COMMAND 3
#define CALLFORMASTER_COMMAND 4
#define AREYOUALIVE_COMMAND 5
#define IAMALIVE_COMMAND 6
#define SLAVEABORT_COMMAND 7
#define ALREADYHAVE_COMMAND 8
#define DONTHAVE_COMMAND 9
#define STARTRENDERING_COMMAND 10
#define RENDERINGDONE_COMMAND 11
#define SLAVEQUIT_COMMAND 13
#define SLAVESTOP_COMMAND 14
#define RENDERINGSTOPPED_COMMAND 15
#define RENDERINGFAILED_COMMAND 16
#define SLAVEKILL_COMMAND 17
```

Fanno parte dei messaggi del protocollo di rendering i seguenti comandi:
`GETFILES_COMMAND, ALREADY_HAVE, DONTHAVE_COMMAND, STARTRENDERING_COMMAND, RENDERINGDONE_COMMAND, RENDERINGFAILED_COMMAND, RENDERINGSTOPPED_COMMAND`.
Tutti gli altri sono messaggi di servizio.

Vediamo ora le fasi del protocollo di comunicazione:

*** Fase di Registrazione dello/degli slaves presso il coordinatore**

Può avvenire in qualunque momento, il Coordinatore è sempre pronto ad accogliere un nuovo slave,anche durante le fasi del protocollo di rendering. Questa azione si esplicita nel modo seguente:

0a. (opzionale) il Coordinatore può inviare un broadcast in rete con il comando CALLFORSLAVES_COMMAND.
L'invio di questo messaggio fa sì che lo Slave esegua il successivo punto 1.
`command: CALLFORSLAVES_COMMAND`
`datagram_id: 0`
`data[ ]: -`

0b. (opzionale) lo Slave può inviare un broadcast in rete con il comando CALLFORMASTER_COMMAND.
L'invio di questo comando fa sì che il Coordinatore svolga il punto 0a.
`command: CALLFORMASTER_COMMAND`
`datagram_id: 0`
`data[ ]: -`

1. uno slave invia al Coordinatore (può avere preso il suo indirizzo dal messaggio di broadcast oppure conoscerlo per altri motivi) un messaggio ADDNODE_COMMAND:
`command: ADDNODE_COMMAND`
`datagram_id: 0`
`data[ ]:` nei primi 4 bytes viene copiato il numero intero corrispondente ai pixels/sec stimati dallo Slave

2. lo slave si porta nella fase iniziale del protocollo

*** Fase di notificazione di indisponibilità dello slave**

Può avvenire in qualunque momento, il Coordinatore è sempre pronto a gestire la situazione

1.Lo Slave invia un messaggio SLAVEABORT_COMMAND al Coordinatore

```
command: SLAVEABORT_COMMAND
datagram_id: 0
data[ ]: -
```

2. Lo Slave può terminare o tornare allo stato iniziale in cui cerca un Coordinatore.

*** Controllo di disponibilità**

Può richiederlo sia lo Slave nei confronti del Coordinatore che viceversa. Viene richiesto dopo un time-out su un altro comando per confermare l'assenza del pari.

1. Il nodo A invia un messaggio AREYOUALIVE_COMMAND al nodo B e si mette in attesa di risposta

```
command: AREYOUALIVE_COMMAND
datagram_id: 0
data[ ]: -
```

2. Il nodo B (se può) risponde con un messaggio IAMALIVE_COMMAND al nodo A

```
command: IAMALIVE_COMMAND
datagram_id: 0
data[ ]: -
```

3. Se il nodo A riceve risposta tutto ok, altimenti se scatta il timeout agisco di conseguenza (chiudo la connessione con il pari, segnalandoglielo (non si sa mai, potrebbe essere ancora in ascolto ma non rispondere)

*** Protocollo di Rendering**

Il Coordinatore deve avere almeno un job pendente, e lo Slave deve essersi registrato presso il coordinatore, cioè deve trovarsi nella fase iniziale del protocollo di rendering.

1. Il Coordinatore invia un messaggio GETFILES_COMMAND allo Slave, in cui indica quali sono i files necessari per il rendering.

```
command: GETFILES_COMMAND
datagram_id: n (n è un identificatore univoco del messaggio, inizialmente vale 0, ad ogni passo è incrementato)
data[ ]: i primi 4 bytes contengono un identificatore univoco della richiesta in esecuzione, i bytes dal quinto in
         poi contengono una sequenza di triple così definite: {A, B, 0} dove A è un intero che indica la
         dimensione del file da trasferire, B è un array di caratteri contenenti il nome del file, 0 è un byte nullo
         che termina la stringa. L'ultima sequenza è terminata da due simboli 0.
```

2. Lo Slave controlla l'indice della richiesta, possono presentarsi due casi distinti:
   a. la richiesta relativa a questo job è già stata servita in precedenza, dunque non sarà necessario scaricare i files
      a1) Lo Slave invia un comando di ALREADYHAVE_COMMAND:
      ```
      command: ALREAYHAVE_COMMAND
      datagram_id: n
      data[ ]: -
      ```
      a2) Lo Slave si mette in attesa sulla socket stream legata alla porta SLAVE_TCP_SOCK eseguendo una
      ```
      accept
      ```
   b. la richiesta relativa a questo job non è mai stata servita da questo slave
      b1) Lo Slave crea una directory temporanea .\request_id\ in cui copierà i files
      b2) Lo Slave invia un comando di DONTHAVE_COMMAND:
      ```
      command: DONTHAVE_COMMAND
      datagram_id: n
      data[ ]: -
      ```
      b3) Lo Slave si mette in attesa sulla socket stream legata alla porta SLAVE_TCP_SOCK eseguendo una
      ```
      accept
      ```
      b4) Il Coordinatore connette la sua socket stream alla porta SLAVE_TCP_SOCK dello Slave eseguendo una
      ```
      connect
      ```
      b5) Il Coordinatore comincia a spedire i files necessari uno dopo l'altro sulla socket stream
      ```
      send
      ```
      b6) Lo Slave legge i dati relativi ai files dalla socket
      ```
      recv
      ```

3. Il Coordinatore esegue una shutdown in scrittura sulla socket stream

```
shutdown
```

4. Lo Slave esegue una shutdown in lettura sulla socket stream;
   `shutdown`
5. Il Coordinatore invia un comando STARTRENDERING_COMMAND, contenente le opzioni per il rendering.
   `command: STARTRENDERING_COMMAND`
   `datagram_id: n+1`
   `data[ ]:` una stringa contenente la linea di comando per il renderer, ad esempio:
   `"+Inomefile.POV +W640 +H480 +AA0.3"`
6. Il coordinatore si mette in attesa dei dati relativi alla bitmap sulla socket stream
   `recv`
7. Lo Slave attiva il rendering e spedisce la bitmap
   `send`
8. Lo Slave chiude la connessione
   `closesocket`
9. Il Coordinatore chiude la connessione
   `closesocket`
10. Lo Slave invia un messaggio per indicare lo stato di terminazione del processo di rendering. Sussistono questi tre casi:
    a. Rendering eseguito correttamente
       `command: RENDERINGDONE_COMMAND`
       `datagram_id: n+1`
       `data[ ]: -`
    b. Rendering terminato in modo anomalo, ad esempio per motivi interni (errori nel file .POV) o esterni (errori sulle socket, o in fase di comunicazione) o per terminazione forzata dello slave
       `command: RENDERINGFAILED_COMMAND`
       `datagram_id: n+1`
       `data[ ]: -`
    c. Rendering terminato su richiesta del Coordinatore, riferendogli l'ultima linea renderizzata con successo
       `command: RENDERINGSTOPPED_COMMAND`
       `datagram_id: n+1`
       `data[ ]:` i primi 4 bytes contengono un intero che indica l'ultima linea tracciata completamente

*** Interruzione di un job per riallocarlo**

Questa azione si inserisce nell'ambito del protocollo precedente, in modo particolare a partire dalla fase 7.
Quando lo Slave si trova nella fase 7. il Coordinatore invia un messaggio SLAVESTOP_COMMAND

```
command: SLAVESTOP_COMMAND
datagram_id: 0
data[ ]: -
```

*** Congedo di uno Slave da parte del Coordinatore**

Questa azione libera uno slave che si era registrato presso un Coordinatore. Ora lo Slave liberato torna in attesa di un coordinatore (in pratica aspetta un comando CALLFORSLAVES_COMMAND se non conosce l'indirizzo del nuovo coordinatore). Questa azione viene eseguita dal Coordinatore anche ogni volta che rileva un suo fallimento o errore di protocollo, o quando termina (in questo caso è opportuno lasciare liberi gli slaves affinchè possano essere usati da altri coordinatori).

L'azione si esplicita nel seguente modo: il Coordinatore invia allo Slave coinvolto un messaggio SLAVEQUIT_COMMAND o un comando SLAVEKILL_COMMAND. I comandi sono equivalenti, ma il secondo oltre a liberare lo Slave ne forza anche la terminazione

```
command: SLAVEQUIT_COMMAND oppure SLAVEKILL_COMMAND
datagram_id: 0
data[ ]: -
```

*** Segnalazione di fallimento di uno slave**

Questa azione può avvenire in qualunque momento, indica al Coordinatore che lo Slave ha fallito, e che non può più fare affidamento su di lui. Il Coordinatore dovrà prenderne atto marcando il job come fallito (pendente) ed eliminando lo slave dalla relativa Tabella. Se il fallimento dello Slave non è grave questo si riporterà nella fase di ricerca di un Coordinatore.

Per effettuare questa segnalazione lo Slave invia al coordinatore il messaggio SLAVEABORT_COMMAND

```
command: SLAVEABORT_COMMAND
datagram_id: 0
data[ ]: -
```

Tolleranza ai guasti

Il progetto presentato, presenta un approccio alla tolleranza ai guasti insito nel protocollo. Ogni nodo in gioco nella comunicazione presta grande attenzione a rilevare fallimenti, e a comunicarli al suo interlocutore, in modo che questo possa effettuare le giuste azioni correttive.

Questa tecnica, anche se mediante sessioni di debugging di simulazione di guasti nelle varie fasi del protocollo ha fornito risultati soddisfacenti, non è tuttavia sufficiente a garantire una piena rilevazione e tolleranza ai guasti. Infatti una terminazione improvvisa e brutale di un thread, oppure un guasto o una partizione della rete riescono a eludere questo approccio.

Per ovviare a questa problematica sono state introdotte diverse tecniche nei soggetti implicati nella comunicazione.

Fondamentalmente i soggetti impegnati nella comunicazione, senza distinzioni tra Slaves e Coordinatore, sono dei threads. In ogni thread che comunica sono stati utilizzati questi accorgimenti per rendere affidabile la comunicazione:

1. ogni thread memorizza l'ultimo messaggio ricevuto, e l'ultimo messaggio inviato, e nelle fasi del protocollo i datagrammi vengono numerati in sequenza.

2. controllo sulla numerazione dei pacchetti. Ad esempio, supponendo che l'ultimo pacchetto arrivato sia quello marcato con il numero *n*, io dovrò rispondere con un datagramma numerato *n+1* o *n* (a seconda che io sia Coordinatore o Slave). Il successivo messaggio dovrà essere per tutti numerato con *n+1*. Effettuando un controllo sulla numerazione dei pacchetti in arrivo, se vedo che il datagram_id del successivo pacchetto in arrivo è minore di n+1, rispedisco l'ultimo pacchetto inviato. Questo comportamento è proprio del thread di ricezione. Se la numerazione non è corretta non notifico ai thread in attesa l'evento di ricezione, ma tento una ritrasmissione dell'ultimo pacchetto inviato, sperando in una successiva ripresa della comunicazione

3. gestione di timeout sulle primitive bloccanti, e sull'attesa di eventi che devono essere segnalati dal thread di ricezione. Qualora scadesse il time-out, rispedisco ogni volta l'ultimo pacchetto inviato fino ad un massimo di 2-3 re-invii o finchè non ottengo la risposta voluta. Se questa non arriva il thread coinvolto segnala il suo fallimento agli altri e termina

4. Prima di fallire per time-out si tenta un approccio di verifica di disponibilità. Allo scadere del time-out si invia un pacchetto AREYOUALIVE_COMMAND, e si torna in attesa. Se arriva la risposta voluta, continuo, se arriva un messaggio "IAMALIVE_COMMAND" torno ad aspettare. Ripeto questa procedura per 2-3 volte, dopodichè segnalo fallimento.

Si può notare come, nel caso di guasti non rilevati da uno dei due pari, il time-out scateni prima un re-invio dell'ultimo messaggio, e successivamente un fallimento, dopo aver tentato con messaggi di tipo "Are you alive?" "I am alive".

Il timeout è stato implementato mediante la primitiva `select` delle Windows Sockets o mediante il supporto delle funzioni Win32 `WaitForSingleObject/WaitForMultipleObjects.`

Supporto Win32 al multithreading ed alla sincronizzazione

L'ambiente Win32 fornisce un supporto nativo al multithreading, dando la possibilità di creare e gestire agevolmente i threads.
I threads vengono creati con la funzione: CreateThread(), che restituisce un handle al thread appena creato.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,          // pointer to security attributes
  DWORD dwStackSize,                                 // initial thread stack size
  LPTHREAD_START_ROUTINE lpStartAddress,             // pointer to thread function
  LPVOID lpParameter,                                // argument for new thread
  DWORD dwCreationFlags,                             // creation flags
  LPDWORD lpThreadId                                 // pointer to receive thread ID );
```

Il puntatore *lpStartAddress* deve puntare ad una funzione del tipo `DWORD WINAPI MyRoutine(LPVOID lp)`, che indica l'indirizzo iniziale del codice del thread.

Per gestire la sincronizzazione tra i threads, Win32 implementa vari tipi di oggetti: Mutex, Semafori, Eventi.Gli eventi sono oggetti sui quali i threads si sospendono nel caso non siano "*settati*". Non appena un *evento* viene segnalato, tutti i thread in attesa vengono risvegliati. Si accede a questi oggetti tramite la primitiva sospensiva:

```
DWORD WaitForSingleObject(
  HANDLE hHandle,                  // handle to object to wait for
  DWORD dwMilliseconds             // time-out interval in milliseconds );
```

Si può anche specificare l'attesa su un oggetto *thread*, che risulterà segnalato alla sua terminazione.

Il codice dell'applicativo dPOV-Ray fa uso di questi oggetti ed eventi per sincronizzare e sospendere i vari threads e per proteggere da inconsistenze l'accesso alle tabelle globali delle Richieste, dei Jobs, degli Slaves

Una volta acquisito il lock sull'oggetto, ed eseguita la sezione critica, la segnalazione si può effettuare tramite le primitive

```
BOOL ReleaseMutex( HANDLE hHandle );
BOOL SetEvent( HANDLE hHandle );
BOOL ResetEvent( HANDLE hHandle );
```

La libreria Windows Sockets, oltre alle classiche primitve comuni alle Berkeley Sockets, contiene primitive di attesa di eventi su socket omogenee e compatibili con le precedenti A tal proposito sono significative le primitive:

```
WSAEVENT  WSACreateEvent(void);
BOOL WSAResetEvent( WSAEVENT hEvent );
BOOL WSACloseEvent( WSAEVENT hEvent );
DWORD WSAWaitForMultipleEvents( DWORD cEvents, const WSAEVENT FAR *lphEvents,
                        BOOL fWaitAll, DWORD dwTimeOUT, BOOL fAlertable );
```

Tuttavia, per rimanere coerenti con la portabilità delle socket di Berkely si è scelto di non utilizzare queste funzionalità, e di continuare ad utilizzare funzioni più tradizionali come `select`.

Riferimenti

- POV-Ray e il suo codice sorgente sono disponibili all'URL: `http://www.povray.org`

- Informazioni sulla piattaforma Win32 e sulle sue chiamate di sistema sono reperibili all'URL
  `http://msdn.microsoft.com`

- "*The Recursive Ray Tracing Algorithm*", di D.Buck, allegato a questa relazione, contiene un'ampia bibliografia sull'argomento ray-tracing.