Lorenzo De Simone N86/1008

# Sistema di monitoraggio della movimentazione merci

Elaborato previsto per il conseguimento dell'esame di Laboratorio di Sistemi Operativi

A.A. 2012/2013

# Indice

# 1. Descrizione generale del progetto

L'applicazione client/server descritta in questo documento rende possibile gestire uno scalo merci contenenti container.

Si ricorda che entrambe le applicazioni sono scritte in linguaggio C e sono pensate per essere eseguite in ambiente UNIX.

Lo scalo è implementato come una matrice NxN e la sua gestione è compito del server. Ogni client rappresenta un container: quando si connette al server, esso viene inserito in una lista apposita, che contiene tutti i container presenti in posizione [0][0]. Questa posizione iniziale verrà chiamata "ingresso" dello scalo: una volta lasciato l'ingresso non sarà più possibile farvi ritorno. E' sempre possibile per un client connettersi al server. La posizione finale sarà invece chiamata "uscita" dello scalo. In sintesi, l'ingresso e l'uscita sulla matrice conterranno sempre 0, mentre le altre posizioni conterranno 0 se vuote, o il descrittore della socket di comunicazione con un client.

Il server genera un thread apposito per ogni client, che ne gestirà le singole richieste ad ogni turno.

Ogni spostamento può esser effettuato in una cella adiacente.

Una collisione si verifica quando due container occupano la stessa cella dello scalo: in tal caso il server provvederà a mandare ad entrambi i container un messaggio e li eliminerà.

I verranno rimossi dalla matrice anche nel caso in cui essi raggiungano la posizione NxN: in tal caso essi escono regolarmente dallo scalo.

Per evitare al più possibile quest'evenienza, il server manda dei messaggi riguardanti lo stato di ogni client, avvertendoli di eventuali container presenti in celle ad esso adiacenti.

# 2. Guida all'utilizzo

Per prima cosa è necessario compilare via terminale entrambi i file sorgente. Per fare ciò, è necessario spostarsi nella directory contenente il file sorgente desiderato e eseguire i comandi qui riportati.

- Server : **gcc -o server server.c -pthread**
- Client: **gcc -o client client.c -pthread**

I nomi "server" e "client" possono essere modificati dall'utente: negli esempi successivi si useranno questi nomi.

A seconda della distribuzione di Linux utilizzata, verranno stampati a video eventuali warning: la presente versione del software è stata compilata senza alcun warning in ambiente **Oracle Linux** (Red Hat Enterprise Linux Server release 6.4 (Santiago).

Una volta compilati entrambi i file sorgente, è possibile proseguire con l'esecuzione.

E'inoltre interessante notare che è possibile eseguire il client re indirizzando lo standard input su un file di testo contenente i comandi desiderati: il testo deve essere però formattato correttamente, poiché altrimenti le funzioni di controllo del client potrebbero non riconoscere i comandi e produrre quindi effetti indesiderati. La formattazione necessaria è piuttosto semplice: basta scrivere un comando su ogni riga seguito dal carattere di newline '**\n**'. In allegato il file **comandi.txt**, mostra un esempio di corretta formattazione: l'esito di quella serie di comandi permette ad un container (escluse eventuali collisioni) di uscire regolarmente da uno scalo 3x3.

## 2.1 Esecuzione ed uso del server

Per eseguire il lato server dell'applicazione, basta utilizzare il seguente comando da terminale:

- **./server "grandezza scalo" "durata del round in secondi"**

L'applicativo controlla che tutte le seguenti regole vengano rispettate dai parametri in input:

1. I parametri in input devono essere due
2. La grandezza minima dello scalo è due
3. La durata del round minima è uno

Il server, una volta eseguito, stamperà a video un messaggio iniziale e provvederà a gestire automaticamente le richieste che gli verranno inoltrate. Il server è in ascolto sulla porta 5200.

Dopo un messaggio di benvenuto, il server stamperà a video un messaggio ad ogni fine round. Ulteriore funzione a disposizione è la stampa di un piccolo log, contenente le informazioni ricevute dai client appena connessi. Il file **serverlog.txt** mostra un esempio d'esecuzione del server. Il file è stato realizzato reindirizzando da terminale lo standard output su di esso: da ciò si evince che tutte le operazioni su standard output utilizzate sono di tipo non bufferizzato e che la re direzione non crea alcun problema.

## 2.2 Esecuzione ed uso del client

Per eseguire il lato server dell'applicazione, basta utilizzare il seguente comando da terminale:

- **./client "indirizzo IP del server" "numero di porta del server(5200)"**

Qualora si voglia sfruttare la possibilità di dare in input un file di testo contenente i comandi, basterà aggiungere **< "file di testo(comandi.txt)"**

L'applicativo controllerà unicamente che il numero di input sia corretto: il controllo della correttezza dei valori in input è delegata all'utente.

E' ovviamente necessario che il server sia già attivo prima di eseguire il client. A connessione avvenuta, il container verrà inserito nella lista d'ingresso e un messaggio iniziale chiederà all'utente di inserire dei comandi.

La lista di comandi disponibili è riportata di seguito:

- **DX**: Sposta il container a destra.
- **SX**: Sposta il container a sinistra.
- **UP**: Sposta il container di sopra.
- **DN**: Sposta il container di sotto.
- **FM**: Tiene il container fermo (Non viene inviato al server)
- **PO**: Richiede al server la propria posizione nello scalo
- **ST**: Richiede al server le seguenti informazioni:
  - Dimensione dello scalo
  - Intervallo tra due round successivi
  - Numero di container nello scalo
  - Posizione di tutti i container nello scalo
  - Numero di possibili collisioni individuate
  - Numero di collisioni registrate

Qualora l'utente scriva un comando errato, il client non lo accetterà e sarà immediatamente pronto a riceverne un altro. I comandi devono essere correttamente formattati e verranno accettati esclusivamente i comandi precedentemente riportati. E' inoltre possibile scrivere più comandi validi di seguito: il client provvederà a scartare quelli non validi e a inviarne uno per ogni round.  Verranno stampate a video sia le informazioni eventualmente richieste dal comando, sia notifiche sullo stato del container.

Il file **clientlog.txt** mostra un esempio d'esecuzione del server: anch'esso è stato ottenuto mediante re indirizzamento e valgono le stesse considerazioni fatte per il server.

# 3 Protocollo client/server

Per la comunicazione fra client e server vengono utilizzate due socket di tipo TCP e per manipolare i dati in esse contenuti si utilizzano le system call **read** e **write**. Su una **prima** socket, il client scrive i messaggi e riceve il feedback dal server sul loro esito e/o eventuali log contenente le informazioni richieste.

Sulla **seconda** socket, invece, il server scriverà lo stato del container e un thread apposito nel client continuerà a leggerne il contenuto in un ciclo infinito: ciò permette di avere notifiche rapide sullo stato del container.

D'ora in avanti nella documentazione le due socket verranno chiamate **prima** socket e **seconda** socket. Di seguito sono riportati i possibili messaggi che il server manda al client attraverso la seconda socket.

| | |
|---|---|
| **R** | Il round è terminato |
| **C** | Il container è stato coinvolto in una collisione |
| **P** | Il container è in procinto di collisione |
| **F** | Il container è uscito regolarmente dallo scalo |
| **K** | Il container è in una posizione stabile |

Il messaggio **R** viene inviato da un thread del server chiamato **sveglia** a tutti i container che in quel round sono rimasti fermi.

Il messaggio verrà poi gestito da un thread apposito nel client che manderà un segnale al thread principale del client: si andrà più nel dettaglio nella sezione della documentazione legata al client.

Oltre alla matrice principale, ci sono delle strutture dati d'appoggio essenziali per il funzionamento dell'applicazione:

| | |
|---|---|
| **lista_tot** | Lista contenente tutti i container, una variabile che dice se hanno mandato un messaggio al server nel round corrente, e il file descriptor della seconda socket. |
| **lista_in** | Lista contenente tutti i container in posizione [0][0] |

Per garantire la coerenza delle informazioni, si è cercato di gestire anche casi di chiusure anomale da parte dei due endpoint di comunicazione: vedremo come verranno gestite nel dettaglio queste problematiche sia sul lato client che sul lato server.

## 3.1 Protocollo lato server

Il server fa le opportune verifiche sui dati in input e sull'ambiente: eventuali errori vengono visualizzati e l'esecuzione termina. Subito dopo le dovute allocazioni e inizializzazioni delle variabili necessarie, il server stampa a video il messaggio di benvenuto e crea il thread sveglia e rimane in attesa di connessioni da parte di client: in questa fase incomincia la stampa a video dei round che accompagnerà tutta l'esecuzione del server.

Alla connessione di un nuovo client, viene generata la prima socket e la gestione di quel client viene delegata ad un thread gestore.

Il thread gestore stampa quindi a video un log contenente le info che lo riguardano: con queste info a disposizione, viene creata la seconda socket di comunicazione.

Poiché un nuovo container è entrato nello scalo, vengono aggiornate sia **lista_tot** che **lista_in**, acquisendo i dovuti mutex.

Viene quindi scritto sulla seconda socket lo stato del container, indicandogli possibili collisioni con i container adiacenti.

Prima di acquisire il lock per la gestione dello scalo, come prima è stato accennato, viene fatta una verifica sull'endpoint di comunicazione: se infatti la read dalla prima socket restituisce 0, il client è stato chiuso in maniera anomala. Si procede quindi alle dovute procedure di eliminazione del container dallo scalo e alla chiusura del thread gestore.

Se la read è andata a buon fine, il server ha ottenuto il comando da eseguire dal client, e il thread cerca di acquisire il lock per la gestione dello scalo. Dopo aver acquisito il lock, il thread verifica che nello scalo, nella posizione memorizzata nelle variabili **rig** e **col**, ci sia effettivamente il suo socket descriptor: in caso contrario, il container ha subito passivamente una collisione. Si procede quindi, in questo caso, solo alla

liberazione del lock e alla terminazione del thread, in quanto il client interessato da quel container è già stato debitamente avvisato dal container che aveva causato la collisione.

Superato questo ulteriore controllo, il server gestisce effettivamente il comando, e gestisce le richieste del client:

- Nel caso di **ST** e **PO**, il server compone una stringa e la scrive sulla prima socket.
- Nel caso di comandi di spostamento, il server fa queste verifiche:
  1. Verifica che il container non esca dalla matrice.
  2. Qualora il container causi una collisione, modifica lo stato in **C** ed elimina entrambi i container da **lista_tot** e dalla matrice. Viene inoltre stampato a video un messaggio.
  3. Qualora il comando sia **DX** o **SX** e il container sia in posizione iniziale, provvede a rimuoverlo dalla **lista_in** (ciò viene verificato mediante una variabile **flag** apposita).
  4. Qualora il comando sia **DX** o **DN** e il container con quello spostamento finirà nell'uscita, esso modifica lo stato in **FI** e rimuove il container da **lista_tot**. Viene stampato un apposito messaggio a video.
  5. Qualora il comando sia **UP** o **SX**, verifica che il container non torni nell'ingresso.

Per i dettagli del funzionamento tecnico delle singole funzioni e dell'aggiornamento delle singole variabili, si rimanda all'analisi del codice sorgente e ai commenti.

Qualora quindi il container, al termine dell'esecuzione del comando ricevuto da client, sia ancora nello scalo, si rimette in attesa del lock per la gestione dello scalo, che verrà sbloccato dal thread sveglia a fine round.

## 3.2 Protocollo lato client

Per prima cosa il client fa le opportune verifiche sui dati in input e sull'ambiente: eventuali errori vengono visualizzati e l'esecuzione termina. Subito dopo le dovute allocazioni e inizializzazioni delle variabili necessarie, il thread associa i segnali **SIGUSR1** e **SIGUSR2** all' handler **hand**, per la corretta gestione dei messaggi ricevuti dalla seconda socket. Infine, viene creato il thread **sveglia** con le info sulla seconda socket: il thread principale accetta poi i comandi da standard input.

Dopo averne verificato la correttezza, il thread principale scrive sulla prima socket il comando se diverso da **FM**, altrimenti rimane in attesa del segnale di fine round (**SIGUSR1**) da parte del thread **sveglia**, prima di eseguire il prossimo comando.

Il thread **sveglia** legge quindi, in parallelo, dalla seconda socket, sulla quale il server manderà gli aggiornamenti sullo stato del container che verranno gestiti mediante degli opportuni controlli ed, eventualmente, provvederanno alla chiusura del thread.

Ancora una volta, per i dettagli tecnici ed implementativi delle singole funzioni, si rimanda al codice sorgente ed ai relativi commenti.

# 4. Dettagli implementativi

In questa sezione verranno discusse nello specifico delle soluzioni proposte ad alcuni problemi, sia sul lato server che sul lato client.

## 4.1 Dettagli implementativi del server

Per la gestione dell'ingresso, anziché impedire l'accesso ai container che entrano nello scalo quando l'ingresso è occupato, si è pensato di permettere a tutti l'accesso in una singola posizione iniziale. L'ingresso è stato implementato mediante una linked list: in essa vi sono tutti i container nell'ingresso. Da essa viene rimosso un nodo quando un container esce dall'ingresso e occupa una posizione nella matrice.

Si è inoltre cercato di prestare attenzione alle problematiche legate a disconnessioni non previste: questo è il blocco di codice nel server che gestisce disconnessioni anomale da parte di un client. Altri dettagli (come la gestione di SIGPIPE) possono essere studiati esaminando il sorgente integrale.

```
if((read(nwcd,comando,3*sizeof(char)))==0)
    {
      sprintf(curr,"Disconnessione anomala del
client [%d] dall'IP [%s]\n",nwcd,ip);

write(STDERR_FILENO,curr,strlen(curr)*sizeof(char));
      pthread_mutex_lock(&acc_lista_tot);

lista_tot=elimina_lista_tot(lista_tot,nwcd,rig,col);
      pthread_mutex_unlock(&acc_lista_tot);
      if(flag==0)//Verifica se il container sta
uscendo dall'ingresso dello scalo.
      {
        pthread_mutex_lock(&acc_lista_in);
        lista_in=elimina_lista(lista_in,nwcd);
        pthread_mutex_unlock(&acc_lista_in);
      }
      else
      {
        pthread_mutex_lock(&acc_comandi);
        scalo[rig][col]=0;
        pthread_mutex_unlock(&acc_comandi);
      }
      pthread_exit(NULL);
    }
```

```
void *sveglia()
{
  struct nodo_tot* temp;
  char round[20];
  //Numero di round passati. Viene inizializzata a
0.
  int n_round=0;

 do
   {
     sleep(T);
     sprintf(round,"Fine round [%d]\n",++n_round);

write(STDOUT_FILENO,round,strlen(round)*sizeof(char)
);
     pthread_mutex_lock(&acc_lista_tot);
     temp=lista_tot;

//Manda il messaggio F a tutti i client che non
hanno mandato messaggi.
     while(temp!=NULL)
     {
       if(temp->spostato==0)
         write(temp->sock,"R\0",2*sizeof(char));
       temp=temp->link;
     }
     pthread_mutex_unlock(&acc_lista_tot);

//Acquisisce il lock per l'accesso ai comandi,
segnala l'inizio del round e
//rilascia il lock.
     pthread_mutex_lock(&acc_comandi);
     pthread_cond_broadcast(&acc_sveglia);
     pthread_mutex_unlock(&acc_comandi);

//Resetta lo spostamento di tutti i container a 0.
     pthread_mutex_lock(&acc_lista_tot);
     resetta_lista_tot(lista_tot);
     pthread_mutex_unlock(&acc_lista_tot);

   }while(1);

}
```

Un'altra scelta di rilievo è stata l'utilizzo di un thread sveglia creato appositamente per sincronizzare tutti i thread del server.

La sua funzione è quella di aspettare effettivamente **T** secondi e scrivere sulla seconda socket di tutti i container inattivi il messaggio **F**. Dopo aver fatto ciò, il thread libera la condition variable e fa in modo che i vari thread acquisiscano di volta in volta il lock per eseguire i comandi che hanno ricevuto dai loro client.

Ovviamente è suo anche l'incarico di resettare **lista_tot**, poiché a inizio round nessun container si è ancora mosso.

## 4.2 Dettagli implementativi del client

L'aspetto forse più complesso del client, che poteva tranquillamente essere single-thread altrimenti, è la gestione del comando **FM**.

Poichè esso **non** deve essere inviato al server, richiede una sincronizzazione più articolata. Si è data una risposta a quest'esigenza creando una seconda socket e rendendo il client multi-thread: qui si riporta il codice del thread **sveglia** del client, che legge dalla seconda socket gli aggiornamenti di stato del container e li gestisce istantaneamente con opportuni messaggi e a video e segnali.

Con questo approccio, pur dovendo gestire una seconda socket, si hanno aggiornamenti istantanei sullo stato di un container: non sarà necessario mandare un messaggio al server per sapere se si è subita una collisione oppure no, l'avviso sarà immediato.

Si noti inoltre come anche qui ci si sia soffermati sul problema delle disconnessioni impreviste:  in particolar modo, con l'approccio proposto, sullo standard error del server e del client si avranno messaggi piuttosto precisi che indicheranno il problema e informeranno il terminale del server di eventuali comandi non ancora gestiti inviati da un client disconnesso.

Viene riportato alla pagina successiva anche il codice dell'handler **hand**, molto semplice, ma utile a distinguere il segnale di fine round da quello di collisione/uscita.

```
void *sveglia()
{
   int clock;
   char mex[2];

//Accept sulla seconda socket.
   if((clock=accept(new_sd,NULL,NULL))==-1)
   {
     write(STDERR_FILENO,"Errore dell'accept nella sveglia\n",33*sizeof(char));
     exit(-1);
   }
   while(1)
   {
//Read dalla seconda socket: da qui si riceveranno
//aggiornamenti istantanei sullo stato del container.
//Nel caso in cui il client debba terminare la propria esecuzione,
//si eseguono delle close() sulle socket non piÙ necessarie.
     if((read(clock,&mex,2*sizeof(char)))!=0)
     {
       if(strcmp(mex,"R")==0)
         pthread_kill(main_t,SIGUSR1); //Segnale di fine round.
       else if(strcmp(mex,"K")==0)
         write(STDOUT_FILENO,"Il container È in una posizione
stabile\n",41*sizeof(char));
       else if(strcmp(mex,"P")==0)
         write(STDOUT_FILENO,"Il container È in procinto di
collisione\n",42*sizeof(char));
       else if(strcmp(mex,"C")==0)
       {
         write(STDOUT_FILENO,"Il container È stato coinvolto in una
collisione\n",50*sizeof(char));
         close(clock);
//Questa write serve solo a sbloccare la read del thread gestore
//e metterlo in attesa del lock:
//in questo modo il server capirÀ  che non si È trattata di una
//disconnessione anomala ma di una collisione subita
         write(sd,"C\0",2*sizeof(char));
         close(sd);
         pthread_kill(main_t,SIGUSR2);
         pthread_exit(NULL);
       }
       else if(strcmp(mex,"F")==0)
       {
         write(STDOUT_FILENO,"Il container È uscito regolarmente dallo
scalo\n",48*sizeof(char));
         close(clock);
         close(sd);
         pthread_kill(main_t,SIGUSR2);
         pthread_exit(NULL);
       }
     }
     else
     {
//La read restituisce 0 e quindi la socket È broken:
//ciÒ È causato da una terminazione anomala del server.
       write(STDERR_FILENO,"Disconnessione anomala del server\n",34*sizeof(char));
       close(clock);
       close(sd);
       exit(-1);
     }
   }
}

void hand(int segnale)
{
  if(segnale==SIGUSR2)
    exit(-1);
}
```

# 5. Codice Sorgente

## 5.1 Codice sorgente del server

```
//Progetto di Laboratorio di sistemi operativi
//Lorenzo De Simone N86/1008

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include <signal.h>
#include <unistd.h>
#include <arpa/inet.h>

int **scalo;
int N,T,collisioni_poss=0,collisioni_reg=0;
pthread_mutex_t acc_lista_in  = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t acc_lista_tot = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t acc_comandi   = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t acc_sveglia    = PTHREAD_COND_INITIALIZER;

//Struttura del nodo della lista di ingresso.
struct nodo
{
   int    info;
   struct nodo *link;
};

//Struttura del nodo della lista contenente tutti i container
//e se essi si sono spostati nel round corrente oppure no.
struct nodo_tot
{
   int info;
   int spostato;
   int sock;
   struct nodo_tot *link;
};

//Struttura in input al thread gestore.
struct info_client
{
  int sd;
  char ip[16];
};

//Variabili globali delle due liste.
struct nodo* lista_in;
struct nodo_tot* lista_tot;
```

```
//Funzioni per la gestione della lista di ingresso.
struct nodo* inserisci_lista(struct nodo* head,int socket);
struct nodo* elimina_lista(struct nodo* head,int socket);

//Funzioni per la gestione della lista totale
struct nodo_tot* inserisci_lista_tot(struct nodo_tot* head, int
socket,int clock);
struct nodo_tot* elimina_lista_tot(struct nodo_tot* head,int socket,int
rig,int col);
void aggiorna_lista_tot(struct nodo_tot* head,int socket);
void resetta_lista_tot(struct nodo_tot* head);

//Funzione che conta gli elementi contenuti in una lista
//server per ottenere le info da mandare al client in risposta al
//comando ST.
int conta_lista();

//Thread che gestisce singolarmente un client.
void *gestore(void *info);

//Thread che scandisce il passare dei round.
void *sveglia();

//Funzione che analizza le possibili collisioni sulla matrice
//prendendo in input le coordinate del container.
char *pre_collisione(int rig, int col);

//Handler per la gestione del segnale SIGPIPE
void hand();

int main (int argc, char *argv[])
{
  pthread_t tid;
  int sd,cd,i;
  socklen_t clilen;
  char temp[100];
  struct sockaddr_in server,client;
  struct info_client info;

//Gestione del segnale SIGPIPE: È necessario in caso di disconnessione
//anomala del client mentre ha ancora comandi da gestire nel buffer
input.
  signal(SIGPIPE,hand);

//Inizializzazione della struttura sockaddr_in.
  server.sin_family=AF_INET;
  server.sin_port=htons(5200);
  server.sin_addr.s_addr=htonl(INADDR_ANY);

  sd=socket(AF_INET,SOCK_STREAM,0);

//Controllo sul numero di input da linea di comando.
  if(argc<3||argc>3)
  {
```

```
    write(STDERR_FILENO,"Non sono stati passati in input due
parametri\n",46*sizeof(char));
    exit(-1);
  }
//Leggo gli input da linea di comando e li assegno alle variabili N e T.
  sscanf(argv[1],"%i",&N);
  sscanf(argv[2],"%i",&T);

//Controlli sulla validitÀ  degli input da linea di comando.
  if(N<=1)
  {
    write(STDERR_FILENO,"Lo scalo deve essere almeno
2x2\n",32*sizeof(char));
    exit(-1);
  }
  else if(T<=0)
  {
    write(STDERR_FILENO,"Il round deve essere di almeno 1
secondo\n",41*sizeof(char));
    exit(-1);
  }

//Allocazione dinamica della matrice che rappresenta lo scalo.
  scalo=(int**)calloc(N,sizeof(int*));

  for(i=0;i<N;i++)
  {
    scalo[i]=(int*)calloc(N,sizeof(int));
  }

//Verifica di errori sulla connessione.
  if((bind(sd,(struct sockaddr*)&server,(socklen_t)sizeof(server))==-1))
  {
    write(STDERR_FILENO,"\nSi È verificato un errore nell'esecuzione
della bind\n",55*sizeof(char));
    exit(-1);
  }
  else
  {
//Messaggio di benvenuto.
    write(STDOUT_FILENO,"\n||Sistema di monitoraggio della movimentazione
merci||\n\n",57*sizeof(char));
    pthread_create(&tid,NULL,sveglia,NULL);
    do
    {
      if((listen(sd,2))==-1)
      {
        write(STDERR_FILENO,"\nErrore nell'esecuzione della
listen\n",37*sizeof(char));
        exit(-2);
      }
      else
      {
        clilen=sizeof(struct sockaddr);
        if((cd=accept(sd,(struct sockaddr*)&client,&clilen))==-1)
        {
```

```
          write(STDERR_FILENO,"\nErrore nell'esecuzione
dell'accept\n",36*sizeof(char));
          exit(-3);
        }
        else
        {

sprintf(temp,"_____________________________________________\n\nConnessione
stabilita con il client [%d]\n",cd);
          write(STDOUT_FILENO,temp,strlen(temp)*sizeof(char));
//Riempio la struttura da passare al thread gestore
//contenente ip e socket descriptor.
          strcpy(info.ip,inet_ntoa(client.sin_addr));
          info.sd=cd;
          pthread_create(&tid,NULL,gestore,(void*)&info);
        }
      }
    }while(1);
  }
}

void *gestore(void *info)
{
  int l,i,j,nwcd,clock,conta,rig=0,col=0,flag=0;
  char *stato;
  char comando[3],porta[6],ip[16],curr[100],messaggio[10000];
  struct sockaddr_in client;
  struct nodo *temp;
  struct info_client *new_info;

  stato=malloc(sizeof(char)*2);

  new_info= (struct info_client*) info;
  nwcd=new_info->sd;
  strcpy(ip,new_info->ip);

//Leggo la porta su cui connettere la socket secondaria dalla socket
primaria.
  read(nwcd,&porta,6*sizeof(char));

//Riempio la struct client con le info ottenute.
  client.sin_family=AF_INET;
  client.sin_port=htons(atoi(porta));//Porta.
  inet_aton(ip,&client.sin_addr);//IP.
  sprintf(curr,"Porta ricevuta dal client: [%s]\nIP del client:
[%s]\n_____________________________________\n",porta,ip);
  write(STDOUT_FILENO,curr,strlen(curr)*sizeof(char));

  clock=socket(AF_INET,SOCK_STREAM,0);

  if(clock<0)
  {
    write(STDERR_FILENO,"Errore socket\n",14*sizeof(char));
    pthread_exit(NULL);
  }
```

```
  if((connect(clock,(struct sockaddr *) &client,sizeof(client)))<0)
  {
    write(STDERR_FILENO,"Errore di connessione al
client\n",32*sizeof(char));
    pthread_exit(NULL);
  }

  pthread_mutex_lock(&acc_lista_in);
  lista_in=inserisci_lista(lista_in,nwcd);
  pthread_mutex_unlock(&acc_lista_in);
  pthread_mutex_lock(&acc_lista_tot);
  lista_tot=inserisci_lista_tot(lista_tot,nwcd,clock);
  pthread_mutex_unlock(&acc_lista_tot);

  do
  {
//Il server aggiorna lo stato del container.
    stato=pre_collisione(rig,col);
//Scrive K oppure P su socket clock.
    write(clock,stato,2*(sizeof(char)));
//Il server riceve da socket il comando del client.
    if((read(nwcd,comando,3*sizeof(char)))==0)
    {
//Questo blocco di codice serve unicamente nel caso in cui il client sia
//terminato in modo anomalo(es. CTRL+C,shutdown hardware,problemi di
connessione):
//il valore di ritorno 0 della read indica una socket broken.
//Si procede quindi all'eliminazione del container dallo scalo per
//preservare la coerenza dei dati all'interno del server e alla chiusura
del thread gestore.
      sprintf(curr,"Disconnessione anomala del client [%d] dall'IP
[%s]\n",nwcd,ip);
      write(STDERR_FILENO,curr,strlen(curr)*sizeof(char));
      pthread_mutex_lock(&acc_lista_tot);
      lista_tot=elimina_lista_tot(lista_tot,nwcd,rig,col);
      pthread_mutex_unlock(&acc_lista_tot);
      if(flag==0)//Verifica se il container sta uscendo dall'ingresso
dello scalo.
      {
        pthread_mutex_lock(&acc_lista_in);
        lista_in=elimina_lista(lista_in,nwcd);
        pthread_mutex_unlock(&acc_lista_in);
      }
      else
      {
        pthread_mutex_lock(&acc_comandi);
        scalo[rig][col]=0;
        pthread_mutex_unlock(&acc_comandi);
      }
      pthread_exit(NULL);
    }
//Si aggiorna la lista totale segnalando che il client ha mandato un
messaggio al server
//in questo round.
    pthread_mutex_lock(&acc_lista_tot);
    aggiorna_lista_tot(lista_tot,nwcd);
```

```
    pthread_mutex_unlock(&acc_lista_tot);

//il thread cerca di acquisite il lock e attende lo sblocco da parte
della sveglia.
    pthread_mutex_lock(&acc_comandi);
    pthread_cond_wait(&acc_sveglia,&acc_comandi);

//Se il thread ha subito una collisione, si chiude e libera il lock.
    if((nwcd!=scalo[rig][col])&&((rig!=0)||(col!=0)))
    {
      pthread_mutex_unlock(&acc_comandi);
      pthread_exit(NULL);
    }

//Di seguito nei vari rami dell'if c'È il codice relativo al singolo
comando:
//nel caso di collisioni o uscita dallo scalo viene modificata la stringa
stato
//e vengono inviati alla socket clock del client gli opportuni messaggi.
//Vengono inoltre gestiti gli spostamenti non validi con opportuni
messaggi d'errore.
    if(strcmp(comando,"UP")==0)
    {
      if(((rig-1)>=0)&&(((rig-1)!=0)||(col!=0)))
      {
        if(scalo[rig-1][col]!=0)
        {
          sprintf(stato,"C");
          sprintf(curr,"Il container [%d] si È scontrato con il container
[%d]\n",nwcd,scalo[rig-1][col]);
          write(STDOUT_FILENO,curr,strlen(curr)*sizeof(char));
          pthread_mutex_lock(&acc_lista_tot);
          lista_tot=elimina_lista_tot(lista_tot,scalo[rig-1][col],rig-
1,col);
          lista_tot=elimina_lista_tot(lista_tot,nwcd,rig,col);
          pthread_mutex_unlock(&acc_lista_tot);
        }
        else
        {
          scalo[rig][col]=0;
          scalo[--rig][col]=nwcd;
        }
        write(nwcd,"Spostamento effettuato\n",23*sizeof(char));
      }
      else
        write(nwcd,"Il carico non puÒ essere spostato nella locazione di
sopra\n",60*sizeof(char));
    }
    else if (strcmp(comando,"DN")==0)
    {
      if((rig+1)<N)
      {
        if(scalo[rig+1][col]!=0)
        {
          sprintf(stato,"C");
```

```
          sprintf(curr,"Il container [%d] si È scontrato con il container
[%d]\n",nwcd,scalo[rig+1][col]);
          write(STDOUT_FILENO,curr,strlen(curr)*sizeof(char));
          pthread_mutex_lock(&acc_lista_tot);

lista_tot=elimina_lista_tot(lista_tot,scalo[rig+1][col],rig+1,col);
          lista_tot=elimina_lista_tot(lista_tot,nwcd,rig,col);
          pthread_mutex_unlock(&acc_lista_tot);
        }
        else
        {
          scalo[rig][col]=0;
          if(rig+1==N-1&&col==N-1)
          {
            sprintf(stato,"F");
            sprintf(curr,"Il container [%d] È uscito regolarmente dallo
scalo\n",nwcd);
            write(STDOUT_FILENO,curr,strlen(curr)*sizeof(char));
            pthread_mutex_lock(&acc_lista_tot);
            lista_tot=elimina_lista_tot(lista_tot,nwcd,rig+1,col);
            pthread_mutex_unlock(&acc_lista_tot);
          }
          else
            scalo[++rig][col]=nwcd;
        }
        if(flag==0)//Verifica se il container sta uscendo dall'ingresso
dello scalo.
        {
          pthread_mutex_lock(&acc_lista_in);
          lista_in=elimina_lista(lista_in,nwcd);
          pthread_mutex_unlock(&acc_lista_in);
          flag=1;
        }
        write(nwcd,"Spostamento effettuato\n",23*sizeof(char));
      }
      else
        write(nwcd,"Il carico non puÒ essere spostato nella locazione di
sotto\n",60*sizeof(char));
    }
    else if (strcmp(comando,"SX")==0)
    {
      if(((col-1)>=0)&&((col-1)!=0)||(rig!=0)))
      {
        if(scalo[rig][col-1]!=0)
        {
          sprintf(stato,"C");
          sprintf(curr,"Il container [%d] si È scontrato con il container
[%d]\n",nwcd,scalo[rig][col-1]);
          write(STDOUT_FILENO,curr,strlen(curr)*sizeof(char));
          pthread_mutex_lock(&acc_lista_tot);
          lista_tot=elimina_lista_tot(lista_tot,scalo[rig][col-
1],rig,col-1);
          lista_tot=elimina_lista_tot(lista_tot,nwcd,rig,col);
          pthread_mutex_unlock(&acc_lista_tot);
        }
        else
```

```
        {
          scalo[rig][col]=0;
          scalo[rig][--col]=nwcd;
        }
        write(nwcd,"Spostamento effettuato\n",23*sizeof(char));
      }
      else
        write(nwcd,"Il carico non puÒ essere spostato nella locazione di
sinistra\n",63*sizeof(char));
    }
    else if (strcmp(comando,"DX")==0)
    {
      if((col+1)<N)
      {
        if(scalo[rig][col+1]!=0)
        {
          sprintf(stato,"C");
          sprintf(curr,"Il container [%d] si È scontrato con il container
[%d]\n",nwcd,scalo[rig][col+1]);
          write(STDOUT_FILENO,curr,strlen(curr)*sizeof(char));
          pthread_mutex_lock(&acc_lista_tot);

lista_tot=elimina_lista_tot(lista_tot,scalo[rig][col+1],rig,col+1);
          lista_tot=elimina_lista_tot(lista_tot,nwcd,rig,col);
          pthread_mutex_unlock(&acc_lista_tot);
        }
        else
        {
          scalo[rig][col]=0;
          if(rig==N-1&&col+1==N-1)
          {
            sprintf(stato,"F");
            sprintf(curr,"Il container [%d] È uscito regolarmente dallo
scalo\n",nwcd);
            write(STDOUT_FILENO,curr,strlen(curr)*sizeof(char));
            pthread_mutex_lock(&acc_lista_tot);
            lista_tot=elimina_lista_tot(lista_tot,nwcd,rig,col+1);
            pthread_mutex_unlock(&acc_lista_tot);
          }
          else
            scalo[rig][++col]=nwcd;
        }
        if(flag==0)//Verifica se il container sta uscendo dall'ingresso
dello scalo.
        {
          pthread_mutex_lock(&acc_lista_in);
          lista_in=elimina_lista(lista_in,nwcd);
          pthread_mutex_unlock(&acc_lista_in);
          flag=1;
        }
        write(nwcd,"Spostamento effettuato\n",23*sizeof(char));
      }
      else
        write(nwcd,"Il carico non puÒ essere spostato nella locazione di
destra\n",61*sizeof(char));
    }
```

```
    else if (strcmp(comando,"PO")==0)
    {
      sprintf(messaggio,"Container [%d] in
posizione[%d][%d]\n",nwcd,rig,col);
      l=strlen(messaggio);
      messaggio[l]='\0';
      write(nwcd,messaggio,l*sizeof(char));
    }
    else if (strcmp(comando,"ST")==0)
    {
//Dimensione dello scalo.
      sprintf(messaggio,"\nDimensione dello scalo: %d\n",N);

//Intervallo tra due round successivi.
      sprintf(curr,"Intervallo tra due round successivi: %d
secondi\n",T);
      strcat(messaggio,curr);

//Tutti i container in posizione [0][0].
      pthread_mutex_lock(&acc_lista_in);
      conta=0;
      if(lista_in!=NULL)
      {
        sprintf(curr,"Tutti i container in posizione [0][0]:\n");
        strcat(messaggio,curr);
        temp=lista_in;
        while(temp!=NULL)
        {
          sprintf(curr,"Container[%d]\n",temp->info);
          strcat(messaggio,curr);
          conta=conta+1;
          temp=temp->link;
        }
      }
      else
      {
        sprintf(curr,"Nessun container in posizione [0][0]\n");
        strcat(messaggio,curr);
      }

//Posizione di tutti i container nel resto dello scalo.
      collisioni_poss=0;
      if(lista_in!=NULL)
        pre_collisione(0,0);

      pthread_mutex_unlock(&acc_lista_in);
      sprintf(curr,"Posizione di tutti i container nel resto dello scalo:
\n");
      strcat(messaggio,curr);

      for(i=0;i<N;i++)
      {
        for(j=0;j<N;j++)
        {
          if(scalo[i][j]!=0)
          {
```

```
            sprintf(curr,"Container [%d] in posizione:
[%d][%d]\n",scalo[i][j],i,j);
            strcat(messaggio,curr);
            conta=conta+1;
            pre_collisione(i,j);
          }
        }
      }

//Numero di container presenti nello scalo.
      sprintf(curr,"Numero di container presenti nello scalo:
%d\n",conta);
      strcat(messaggio,curr);

//Numero di possibili collisioni individuate.
      sprintf(curr,"Numero di possibili collisioni individuate:
%d\n",collisioni_poss/2);
      strcat(messaggio,curr);

//Numero di collisioni registrate.
      sprintf(curr,"Numero di collisioni registrate:
%d\n\n",collisioni_reg);
      strcat(messaggio,curr);
      write(nwcd,messaggio,strlen(messaggio)*sizeof(char));
    }

//Nel caso in cui lo stato È "C", viene aumentato il contatore delle
collisioni e
//il thread termina dopo aver liberato il lock.
//Nel caso in cui lo stato È "F", il thread termina e libera il lock.
//Nel caso generale, viene semplicemente liberato il lock.
    if(strcmp(stato,"C")==0)
    {
      collisioni_reg=collisioni_reg+1;
      pthread_mutex_unlock(&acc_comandi);
      pthread_exit(NULL);
    }
    else if(strcmp(stato,"F")==0)
    {
      pthread_mutex_unlock(&acc_comandi);
      pthread_exit(NULL);
    }
    pthread_mutex_unlock(&acc_comandi);

  }while(1);
}

void *sveglia()
{
  struct nodo_tot* temp;
  char round[20];
  //Numero di round passati. Viene inizializzata a 0.
  int n_round=0;

 do
   {
```

```
      sleep(T);
      sprintf(round,"Fine round [%d]\n",++n_round);
      write(STDOUT_FILENO,round,strlen(round)*sizeof(char));
      pthread_mutex_lock(&acc_lista_tot);
      temp=lista_tot;

//Manda il messaggio F a tutti i client che non hanno mandato messaggi.
      while(temp!=NULL)
      {
        if(temp->spostato==0)
          write(temp->sock,"R\0",2*sizeof(char));
        temp=temp->link;
      }
      pthread_mutex_unlock(&acc_lista_tot);

//Acquisisce il lock per l'accesso ai comandi, segnala l'inizio del round
e
//rilascia il lock.
      pthread_mutex_lock(&acc_comandi);
      pthread_cond_broadcast(&acc_sveglia);
      pthread_mutex_unlock(&acc_comandi);

//Resetta lo spostamento di tutti i container a 0.
      pthread_mutex_lock(&acc_lista_tot);
      resetta_lista_tot(lista_tot);
      pthread_mutex_unlock(&acc_lista_tot);

    }while(1);

}

//Inserisce un nuovo container nella lista d'ingresso.
struct nodo* inserisci_lista(struct nodo* head, int socket)
{
  struct nodo *nuovo,*temp;
  nuovo=malloc(sizeof(struct nodo));
  nuovo->info=socket;
  nuovo->link=NULL;

  if(head==NULL)
    head=nuovo;
  else
  {
    temp=head;
    while(temp->link!=NULL)
    {
      temp=temp->link;
    }
    temp->link=nuovo;
  }
  return(head);
}

//Elimina un nodo dalla lista d'ingresso.
struct nodo* elimina_lista(struct nodo* head,int socket)
{
```

```
  struct nodo *temp,*prec;
  temp=head;
  prec=temp;

  while((temp!=NULL)&&(temp->info!=socket))
  {
   prec=temp;
   temp=temp->link;
  }
  if(head->info==socket)
    head=head->link;
  else
    prec->link=temp->link;

  free(temp);
  return(head);
}

//Controlla se ci sono container nelle zone adiacenti
//e restituisce il nuovo stato.
char *pre_collisione(int rig, int col)
{
  char *stato;
  stato=malloc(2*(sizeof(char)));
  sprintf(stato,"K");

  if(col!=N-1)//Controllo a destra(DX).
  {
    if(scalo[rig][col+1]!=0)
    {
      sprintf(stato,"P");
      collisioni_poss=collisioni_poss+1;
    }
  }

  if(rig!=N-1)//Controllo giÀ¹(DN).
  {
    if(scalo[rig+1][col]!=0)
    {
      sprintf(stato,"P");
      collisioni_poss=collisioni_poss+1;
    }
  }

  if(rig-1>=0)//Controllo su(UP).
  {
    if((scalo[rig-1][col]!=0)||((rig-1==0&&col==0)&&(lista_in!=NULL)))
    {
      sprintf(stato,"P");
      collisioni_poss=collisioni_poss+1;
    }
  }

  if(col-1>=0)//Controllo a sinistra(SX).
  {
    if((scalo[rig][col-1]!=0)||((rig==0&&col-1==0)&&(lista_in!=NULL)))
```

```
    {
      sprintf(stato,"P");
      collisioni_poss=collisioni_poss+1;
    }
  }
  return stato;
}

//Inserisce un nuovo nodo nella lista totale.
struct nodo_tot* inserisci_lista_tot(struct nodo_tot* head,int socket,int
clock)
{
  struct nodo_tot *nuovo,*temp;

  nuovo=malloc(sizeof(struct nodo_tot));
  nuovo->info=socket;
  nuovo->link=NULL;
  nuovo->spostato=0;
  nuovo->sock=clock;

  if(head==NULL)
    head=nuovo;
  else
  {
    temp=head;
    while(temp->link!=NULL)
    {
      temp=temp->link;
    }
    temp->link=nuovo;
  }
  return(head);
}

//Elimina un nodo dalla lista totale e scrive l'avvenuta collisione sul
socket clock del container.
//Viene inoltre eseguita una close() su entrambe le socket legate al
container.
struct nodo_tot* elimina_lista_tot(struct nodo_tot* head,int socket,int
rig, int col)
{
  struct nodo_tot *temp,*prec;

  temp=head;
  prec=temp;

  while((temp!=NULL)&&(temp->info!=socket))
  {
   prec=temp;
   temp=temp->link;
  }

  if(temp!=NULL)
  {
    if((rig==N-1)&&(col==N-1))
      write(temp->sock,"F\0",2*sizeof(char));
```

```
    else
      write(temp->sock,"C\0",2*sizeof(char));

    close(temp->info);
    close(temp->sock);
  }

  if((head!=NULL)&&(head->info==socket))
    head=head->link;
  else if(temp!=NULL)
  {
    prec->link=temp->link;
    free(temp);
  }

  scalo[rig][col]=0;

  return(head);
}

//Setta ad 1 lo spostamento del container dato in input.
void aggiorna_lista_tot(struct nodo_tot* head,int socket)
{
  struct nodo_tot *temp;
  temp=head;

  while((temp!=NULL)&&(temp->info!=socket))
  {
   temp=temp->link;
  }
  if(temp!=NULL)
    temp->spostato=1;
}

//Resetta tutta gli spostamenti dei container settandoli a 0.
void resetta_lista_tot(struct nodo_tot* head)
{
   struct nodo_tot* temp;
   temp=head;

   while(temp!=NULL)
   {
     temp->spostato=0;
     temp=temp->link;
   }
}

//Handler utile nel remoto caso in cui il client si disconnette in modo
anomalo
//mentre ha ancora comandi da gestire nel buffer input.
void hand ()
{
  write(STDERR_FILENO,"Alcuni comandi del client non sono stati
gestiti\n",49*sizeof(char));
}
```

## 5.2 Codice sorgente del client

```
//Progetto di Laboratorio di sistemi operativi
//Lorenzo De Simone N86/1008

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#include <arpa/inet.h>

//Funzione che controlla la validitÀ  dei comandi.
int controllo(char *cmd);

//Funzione sveglia.
void *sveglia();

//Handler per il messaggio di fine round.
void hand(int segnale);

//Socket descriptor della socket per i messaggi.
int sd;

//Socket descriptor della socket per la sveglia.
int new_sd;

//Tid del thread principale.
pthread_t main_t;

int main(int argc, char *argv[])
{
  int l,test;
  char comando[3],porta[6],messaggio[10000];
  struct sockaddr_in server,client;
  socklen_t len;
  pthread_t tid;

//Controllo sul numero di input da linea di comando.
  if(argc<3||argc>3)
  {
    write(STDERR_FILENO,"Non sono stati passati in input due
parametri\n",46*sizeof(char));
    exit(-1);
  }
//Assegno alla variabile main il valore del tid del thread principale.
  main_t=pthread_self();
//L'handler hand viene associato al segnale SIGUSR1 e al segnale SIGUSR2.
  signal(SIGUSR1,hand);
  signal(SIGUSR2,hand);
```

```
//Inizializzazione della struttura sockaddr_in.
  server.sin_family=AF_INET;
  server.sin_port=htons(atoi(argv[2]));
  inet_aton(argv[1],&server.sin_addr);
  sd=socket(AF_INET,SOCK_STREAM,0);

  if(connect(sd,(struct sockaddr *)&server,(socklen_t) sizeof(server))==-
1)
  {
     write(STDERR_FILENO,"\nErrore di connessione al
server\n",33*sizeof(char));
     exit(-1);
  }
  else
  {
//Prima Connessione avvenuta con successo.
//Creazione della seconda socket per la sveglia.

    new_sd=socket(AF_INET,SOCK_STREAM,0);
    if(new_sd<0)
    {
       write(STDERR_FILENO,"Errore nella creazione della seconda
socket\n",44*sizeof(char));
       exit(-1);
    }

    len=(socklen_t)sizeof(client);
    listen(new_sd,1);

//Niente bind: verrÀ  assegnata una porta effimera.
    if(getsockname(new_sd,(struct sockaddr*) &client, &len)<0)
    {
      write(STDERR_FILENO,"Errore recupero numero
porta\n",29*sizeof(char));
      exit(-1);
    }
//Stringa con la porta.
    sprintf(porta,"%d",ntohs(client.sin_port));

//Scrittura  della porta su socket.
    write(sd,&porta,(strlen(porta)+1)*sizeof(char));

//Creo il thread sveglia.
   pthread_create(&tid,NULL,sveglia,NULL);

    write(STDOUT_FILENO,"\nInserire i comandi da inviare al
server\n",41*sizeof(char));
    do
    {
      do
      {
        l=read(STDIN_FILENO,comando,3*sizeof(char));//Lettura del comando
da standard input.
        if(comando[l-1]=='\n')
          comando[l-1]='\0';
```

```
        else
          while(getchar()!='\n'); //Pulisce il buffer della tastiera in
caso di input errato.
//Legenda dei valori assumibili dalla funzione controllo():
//0=Valido e !=FM
//1=Non valido
//2=FM
        test=controllo(comando);
      }while(test==1);

      if(test==0)//Comando da inviare al server.
      {
        write(sd,comando,l*sizeof(char));
        l=read(sd,messaggio,10000*sizeof(char));
        if(l!=0)
          write(STDOUT_FILENO,messaggio,l*sizeof(char));
      }
      else if(test==2)//Comando FM.
      {
        pause();//Attesa di SIGUSR1 dalla sveglia.
        write(STDOUT_FILENO,"Fine round\n",11*sizeof(char));
      }
    }while(1);
  }
}

//Viene controllata la validitÀ  del comando in input.
int controllo(char *cmd)
{
  if (strcmp(cmd,"UP")!=0)
    if (strcmp(cmd,"DN")!=0)
      if (strcmp(cmd,"SX")!=0)
        if (strcmp(cmd,"DX")!=0)
          if (strcmp(cmd,"ST")!=0)
           if (strcmp(cmd,"PO")!=0)
             if (strcmp(cmd,"FM")!=0)
             {
               write(STDOUT_FILENO,"Comando non
riconosciuto\n",25*sizeof(char));
               return(1);//Comando non valido.
             }
             else
               return(2);//Comando FM.

return(0);//Comando valido != FM.
}

void *sveglia()
{
   int clock;
   char mex[2];

//Accept sulla seconda socket.
   if((clock=accept(new_sd,NULL,NULL))==-1)
   {
```

```
     write(STDERR_FILENO,"Errore dell'accept nella
sveglia\n",33*sizeof(char));
     exit(-1);
   }

   while(1)
   {
//Read dalla seconda socket: da qui si riceveranno
//aggiornamenti istantanei sullo stato del container.
//Nel caso in cui il client debba terminare la propria esecuzione,
//si eseguono delle close() sulle socket non piÀ¹ necessarie.
     if((read(clock,&mex,2*sizeof(char)))!=0)
     {
       if(strcmp(mex,"R")==0)
         pthread_kill(main_t,SIGUSR1); //Segnale di fine round.
       else if(strcmp(mex,"K")==0)
         write(STDOUT_FILENO,"Il container È in una posizione
stabile\n",41*sizeof(char));
       else if(strcmp(mex,"P")==0)
         write(STDOUT_FILENO,"Il container È in procinto di
collisione\n",42*sizeof(char));
       else if(strcmp(mex,"C")==0)
       {
         write(STDOUT_FILENO,"Il container È stato coinvolto in una
collisione\n",50*sizeof(char));
         close(clock);
//Questa write serve solo a sbloccare la read del thread gestore
//e metterlo in attesa del lock:
//in questo modo il server capirÀ  che non si È trattata di una
//disconnessione anomala ma di una collisione subita
         write(sd,"C\0",2*sizeof(char));
         close(sd);
         pthread_kill(main_t,SIGUSR2);
         pthread_exit(NULL);
       }
       else if(strcmp(mex,"F")==0)
       {
         write(STDOUT_FILENO,"Il container È uscito regolarmente dallo
scalo\n",48*sizeof(char));
         close(clock);
         close(sd);
         pthread_kill(main_t,SIGUSR2);
         pthread_exit(NULL);
       }
     }
     else
     {
//La read restituisce 0 e quindi la socket È broken:
//ciÒ È causato da una terminazione anomala del server.
       write(STDERR_FILENO,"Disconnessione anomala del
server\n",34*sizeof(char));
       close(clock);
       close(sd);
       exit(-1);
     }
   }
```

```
}

void hand(int segnale)
{
  if(segnale==SIGUSR2)
    exit(-1);
}
```