TESINA DI SISTEMI OPERATIVI

A.A.2019/2020

Studente Matteo Ciccaglione

Matricola 0266643

Indice

Introduzione

Specifica della tesina

Sistema di prenotazione posti

Realizzazione di un servizio di prenotazione posti per una sala cinematografica, supportato da un server.

Ciascun posto è caratterizzato da un numero di fila, un numero di poltrona, e può essere libero o occupato. Il server accetta e processa sequenzialmente o in concorrenza (a scelta) le richieste di prenotazione di posti dei client (residenti, in generale, su macchine diverse).

Un client deve fornire ad un utente le seguenti funzioni:

1. Visualizzare la mappa dei posti in modo da individuare quelli ancora disponibili.
2. Inviare al server l'elenco dei posti che si intende prenotare (ciascun posto da prenotare viene ancora identificato tramite numero di fila e numero di poltrona).
3. Attendere dal server la conferma di effettuata prenotazione ed un codice univoco di prenotazione.
4. Disdire una prenotazione per cui si possiede un codice.

Descrizione dell'applicazione server

In questa sezione si esegue una analisi puramente qualitativa dell'applicazione server e le operazioni offerte all'utente finale.

Il server è stato realizzato con il fine di gestire e amministrare le prenotazioni relative ad una sala del cinema. Il server offre due operazioni all'utente finale che lo utilizza:

1) Operazione status: questa operazione consente di visualizzare in qualsiasi momento lo stato della sala di proiezione cinema e l'insieme delle prenotazioni attive, ciascuna prenotazione è rappresentata da un codice di prenotazione univoco e una lista di coppie del tipo row,col rappresentante rispettivamente la fila e il posto in fila assegnato a quella prenotazione.
2) Operazione quit: questa operazione consente di chiudere correttamente il server.

Descrizione dell'applicazione client

L'applicazione client offre all'utente finale una interfaccia molto semplice: Una mappa della sala cinema così strutturata: la mappa ha l'aspetto di una matrice avente tante righe quante sono le file presenti nella sala cinema, in ciascuna riga la prima colonna indica il numero di fila mentre le restanti colonne il numero di poltrona in quella fila.

L'utente può operare con il client tramite 3 operazioni identificate tramite un codice numerico:

Codice 0: si richiede di effettuare una prenotazione per uno o più posti della sala cinema. Ciascun posto dovrà essere inserito rispettando la seguente sintassi: numero_fila, numero_poltrona. Le differenti richieste devono essere separate da uno spazio vuoto, ad esempio supponiamo di voler prenotare i posti in fila 1 numero di poltrona 2,3,4 allora la richiesta sarà del tipo 1,2 1,3 1,4. Qualsiasi input in formato differente sarà rifiutato. In caso di avvenuta prenotazione verrà consegnato un codice di prenotazione unico.

Codice 1: si richiede di disdire una prenotazione identificata da un codice di prenotazione. L'utente dovrà inserire il codice di prenotazione associato per poterla disdire.

Codice 2: si richiede la corretta chiusura dell'applicazione.

L'avvio dell'applicazione client richiede all'utente di inserire la coppia IP PortNumber del server al quale intende connettersi.

Specifiche tecniche

L’applicazione server

L’applicativo server è stato progettato ed implementato come applicazione multithread in grado di accettare e gestire connessioni multiple di tipo AF_INET.

Per garantire l’affidabilità e persistenza delle informazioni relative ai codici il server gestisce un file di backup chiamato “server_backup_file”. Il file è così strutturato: ogni riga del file (sequenza di caratteri terminata da uno ‘\n’) è una stringa di numeri separati dal carattere speciale ‘#’, dove il primo valore rappresenta il codice univoco di prenotazione e i successivi i posti prenotati.

I posti vengono identificati dal server tramite un numero ottenuto dalla seguente espressione matematica:

(n_fila-1)*MAX_POSTI_IN_FILA + poltrona-1 (1)

Dove n_fila e poltrona rappresentano rispettivamente il numero di fila e il numero di poltrona richiesto dal cliente mentre MAX_POSTI_IN_FILA è il numero massimo di poltrone disponibili in una file della sala. Per esempio supponendo una sala con un massimo di 8 posti per fila allora il valore 3 corrisponderà alla prenotazione fila 1 poltrona 4. Al lancio il main thread crea e lancia un thread (d’ora in avanti il backup thread) responsabile della gestione del file di backup.

La richiesta di un codice comporta la scrittura di quest’ultimo e della prenotazione associata in coda al file di backup. Per quanto concerne la cancellazione della prenotazione associata ad un codice anche questa comporta la scrittura di una riga in fondo al file contenente unicamente il codice associato alla prenotazione. Sarà responsabilità del backup thread riorganizzare il file rimuovendo le righe corrispondenti a prenotazioni non più valide. La funzione eseguita periodicamente (ad intervalli temporali di 300 secondi) dal thread cancella il contenuto attuale del file sostituendolo con le sole prenotazioni attualmente valide. Tale procedura è eseguita in totale isolamento rispetto agli altri thread che richiedono l’accesso al file, da cui la scelta di introdurre un delay di 300 secondi tra una esecuzione e la successiva. Non potendo garantire lo stato corretto del file di backup al momento della chiusura dell’applicazione (la quale può avvenire sia a causa di un malfunzionamento che per volontà dell’utente) la funzione di inizializzazione del server eseguirà come prima procedura il riordino del contenuto del file di backup, per poi procedere all’inizializzazione delle strutture dati gestite a livello applicativo.

La sala cinema è gestita dal server come un array di MAX_FILA*MAX_POSTI_IN_FILA di interi. Ogni posto è indicizzato secondo (1) e il valore associato è 1 se il posto è libero 0 altrimenti.

Il main thread crea una socket associata all’indirizzo ip della macchina che ospita il server e il port number indicato dall’utente (25001 di default). La socket viene posta in stato listening e attraverso un loop infinito viene utilizzata per ricevere ed accettare connessioni in ingresso da parte dei client. Ogni connessione viene gestita da un thread specifico creato al momento dell’avvenuta connessione. Ciascun thread è responsabile di ricevere tradurre e processare le richieste relative al

proprio client. L'operazione di processamento della richiesta del client avviene in totale isolamento rispetto agli altri thread in quanto il processamento di una richiesta richiede tipicamente l'accesso a diverse strutture condivise quali ad esempio l'array della sala, l'array dei codici, la coda dei messaggi in uscita.

I messaggi prodotti da una operazione di processamento vengono inseriti all'interno di una coda di messaggi gestita da un altro thread (d'ora in avanti il sender_thread) il quale è responsabile dell'invio dei messaggi presenti in coda verso i client interessati. In particolare il server mantiene due differenti code di messaggio implementate tramite una lista collegata di "msg", dove msg è un tipo di dato definito dalla seguente struttura in c

```
struct __message{
        int type; // 1=assegnazione posti, 2=disdire operazione, 3=gestione
        char *text;//contenuto del messaggio
        int sock_dest; //unused for type 1 and 2. Se type=3 e il messaggio lo invia il server allora sock_dest è il socket destinazione.
}
```

Le liste hanno un differente livello di priorità. I messaggi aventi type=3 vengono inseriti nella lista a priorità maggiore, in quanto destinati a conservare informazioni quali codici di prenotazione o messaggi di gestione che si vuole consegnare rapidamente al client interessato. I messaggi di tipo 3 hanno un unico destinatario, mentre gli altri messaggi vengono inoltrati verso tutti i client connessi in quanto rappresentano una variazione nello stato della sala. In questo modo si garantisce che ciascun client connesso al server operi sempre sullo stesso stato della mappa dei posti. Il sender thread invia sempre prima un messaggio in coda ad alta priorità se presente, facendolo poi seguire da un messaggio a bassa priorità.

L'applicativo tiene traccia della socket attualmente in uso in operazioni di scrittura tramite una variabile globale, in questo modo il gestore di SIGPIPE si limita alla richiesta di rimozione della socket non più raggiungibile.

L'applicazione client

Anche il client come il server è un'applicazione multithread, in particolare il client presenta 2 thread: il main thread responsabile dell'inizializzazione delle strutture utilizzate e della richiesta di connessione al server, compresa la gestione dell'input da parte dell'utente. A tal fine è stato introdotto un delay di 10 secondi in seguito a ciascuna operazione portata a termine dal client per permettere all'utente di prendere visione del risultato riportato. Inoltre il main thread è responsabile dell'invio dei messaggi al server. L'input richiesto all'utente prevede una serie di coppie di valori separati dal " " dove ciascuna coppia è così formata n_fila,n_poltrona come

indicato nella sezione introduttiva. L'input viene poi opportunamente convertito per renderlo conforme ai dati gestiti dal server riportandolo nella forma descritta da (1).

Il client similmente al server rappresenta internamente la sala cinematografica come una stringa dove ciascun carattere è 0 se il posto corrispondente è occupato e 1 altrimenti.

Un secondo thread ha la responsabilità di leggere e processare correttamente tutti i messaggi in arrivo da parte del server e gestire opportunamente la stringa dei posti.

Per evitare di mantenere connessioni inutilizzate verso il server, il client implementa un timer (realizzato tramite l'utilizzo del segnale SIGALARM e della chiamata di sistema alarm) di 500 secondi (circa 8 minuti) allo scadere del quale la connessione verso il server viene automaticamente interrotta e l'applicazione viene terminata. Il timer si resetta automaticamente ogni volta che l'utente interagisce con l'applicazione.

La decisione di terminare l'applicazione in seguito ad una disconnessione è legata a diversi motivi: la disconnessione può avvenire per volere dell'utente o per indisponibilità del server o addirittura per inattività dunque non ho ritenuto opportuno operare un tentativo di riconnessione.

La comunicazione client server

Dal momento che l'applicazione server accetta connessioni provenienti da qualsiasi applicazione (non necessariamente il client) si è deciso di introdurre un "protocollo" di comunicazione tra le due applicazioni (che è tipicamente mascherato agli utenti) per evitare connessioni esterne all'applicazione. Sostanzialmente la comunicazione avviene attraverso messaggi racchiusi tra due "tag".

Tag di apertura: Questo tag indica l'inizio e il tipo del messaggio e può essere di 3 tipi

"-a" indica che il contenuto del messaggio è una serie di stringhe separate da " " contenenti valori numerici corrispondenti a posti della sala secondo la formula introdotta in (1).

"-d" indica che il contenuto del messaggio è un codice numerico corrispondente ad un codice di prenotazione.

"-c" indica che il contenuto del messaggio è un messaggio di gestione.

Ed un tag di chiusura "-f" che indica la terminazione del messaggio corrente.

Richieste in ingresso al server che non rispettano la suddetta sintassi verranno rifiutate e la connessione immediatamente chiusa.

Come già spiegato il tag "-d" indica al server la volontà di rimuovere una prenotazione tramite un codice. Questa procedura genera un messaggio di tipo 3 (vedi discussione sui tipi nella sezione riservata al server) con tag di gestione contenente uno tra i due valori: "ct" se il codice richiesto è

stato trovato nel server e la cancellazione è andata a buon fine, “cnt” se l’operazione richiesta non ha avuto esito positivo perché il codice è inesistente. Il tag “-d” invece indica al client di aggiornare la mappa liberando i posti indicati nel testo che segue.

Il tag “-a” indica al server la volontà di prenotare i posti indicati nel corpo del messaggio. Il processamento di questa operazione produce un messaggio di tipo 3 contenente il tag “-c” il cui corpo del messaggio contiene il codice univoco associato a tale prenotazione. Tale tag indica al client di aggiornare la mappa marcando come riservati i posti indicati nel testo del messaggio.

I messaggi prima di essere processati vengono opportunamente tradotti da una funzione ReadCommand definita (ma implementata diversamente) in entrambi gli applicativi.

Infine discuto l’aspetto relativo all’inizializzazione del client che avviene in seguito alla connessione con il server. Il messaggio di inizializzazione a differenza dei precedenti non utilizza il meccanismo dei tag ma una stringa così formattata “MaxFila,MaxPoltrona,statusSala”. La stringa statusSala corrisponde alla rappresentazione lato client della sala cinematografica.

Codice applicativo e istruzioni per l’installazione

Istruzioni per l’installazione

L’applicazione server è accompagnata da un installer (server_installer.c) che ha il compito di inizializzare la sala cinema che si vuole rappresentare. N.B. Per consentire il corretto funzionamento dell’applicazione server è necessario procedere prima all’installazione tramite installer. All’interno della directory server è presente un Makefile. Con la direttiva make si esegue la compilazione e successivo lancio dell’installer. Segue poi a termine della configurazione la compilazione dell’applicativo server. A questo punto è possibile eseguire il server con il comando ./serverApplication da shell Linux. Una successiva installazione cancellerà il contenuto attuale della directory e del server. E’ possibile rimuovere l’installer, il server e tutto il contenuto di gestione tramite “make clear” L’applicazione client presente nella directory client contiene un Makefile che può essere eseguito con il comando make per compilare l’applicativo. A questo punto è possibile eseguire il client con il comando ./client ipNumber portNumber da shell linux. E’ possibile rimuovere il client tramite comando “make clear”

Codice server

Il codice dell’applicativo server si trova in due file server.c e headerS.h. Nel secondo file sono definite ed implementate le funzioni utilizzate dal server. Il primo file contiene le implementazioni delle funzioni eseguite dai thread e i gestori degli eventi. Infine è presente un file describer.h creato dall’installer e contenente le definizioni delle macro NUMROW e NUMCOL come stabilito

all’atto dell’installazione. Il contenuto di questo file è noto solo in seguito all’installazione pertanto non viene riportato.

File server.c

```
#include"headerS.h" //definisce la maggior parte delle funzioni e gli header di libreria
void initialize_server(){
	riordinaFile();
	backup=fopen("server_backup_file.txt","r+");
	int codice=0,i=0;
	int len=0;
	char string[1024],*local;
	for(i=0;i<NUMROW*NUMCOL;i++){
		matrix[i]=1;
		sockets[i]=-1;
	}
	while(fscanf(backup,"%s\n",string)!=EOF){
		len=strlen(string);
		local=strtok(string,"#");
		codice=atoi(local);
		code[codice]=malloc(sizeof(char*)*len);
		strcpy(code[codice],"");
		local=strtok(NULL,"#");
		while(local!=NULL){
			strcat(code[codice],local);
			strcat(code[codice]," ");
			matrix[atoi(local)]=0;
			local=strtok(NULL,"#");
		}
		if(strcmp(code[codice],"")==0){
```

```
                code[codice]=NULL;
            }

        }
        fclose(backup);
}

void initialize_client(int sock){
        char string[100+NUMROW*NUMCOL];
        char str[NUMROW*NUMCOL+1];
        int i,c;
        sprintf(string,"%d,%d,",NUMROW,NUMCOL);
        for(i=0;i<NUMROW;i++){
            for(c=0;c<NUMCOL;c++){
                if(matrix[i*NUMCOL+c]==0){
                    str[i*NUMCOL+c]='0';
                }
                else{
                    str[i*NUMCOL+c]='1';
                }
            }
        }
        str[NUMROW*NUMCOL]='\0';
        strcat(string,str);
        if(write(sock,string,strlen(str))==-1){
        }
        pthread_mutex_lock(&init_mut);
        pthread_mutex_lock(&mutex_socket);
        for(i=0;i<NUMROW*NUMCOL;i++){
            if(sockets[i]==-1){
```

```
                        sockets[i]=sock;
                        if(i>max_client){
                                max_client=i;
                        }
                        break;
                }
        }
        pthread_mutex_unlock(&mutex_socket);
        pthread_mutex_unlock(&init_mut);
}
void *sender_thread(void *Unused){
        msg *mex;
        int num;
        sigset_t set;
        sigaddset(&set,SIGPIPE);
        sigaddset(&set,SIGINT);
        sigprocmask(SIG_UNBLOCK,&set,NULL);
        list *local;
        struct sembuf oper;
        oper.sem_num=0;
        oper.sem_op=-1;
        while(1){
                semop(sem_elem,&oper,1);
                if(highprio_h!=NULL){
                        mex=highprio_h->text;
                        pthread_mutex_lock(&mutex_l);
                        if(highprio_t==highprio_h){
                                highprio_t=NULL;
                        }
                        local=highprio_h;
```

```
                highprio_h=highprio_h->next;
                pthread_mutex_unlock(&mutex_l);
                WriteCommand(mex);
                free(local);
                free(mex);
        }
        if(lowprio_h!=NULL){
                mex=lowprio_h->text;
                pthread_mutex_lock(&mutex_lp);
                if(lowprio_t==lowprio_h){
                        lowprio_t=NULL;
                }
                local=lowprio_h;
                lowprio_h=lowprio_h->next;
                pthread_mutex_unlock(&mutex_lp);
                WriteCommand(mex);
                free(local);
                free(mex);
        }
    }
}



void *backupThread(void *unused){
        while(1){
        sleep(300);
        riordinaFile();
        }
}
```

```
void pipe_handler(int sig){
	printf("SIGPIPE received client disconnected...\n");
	removeSock(actual_sock);
}

void *thread_f(void *sock){
	int my_sock=*(int *)sock;
	sigset_t set;
	sigfillset(&set);
	sigprocmask(SIG_BLOCK,&set,NULL);
	msg *mex=malloc(sizeof(msg));
	while(1){
		mex=ReadCommand(my_sock);
		if(mex==NULL){
			removeSock(my_sock);
			pthread_exit(NULL);
		}
		ProcessCommand(mex,my_sock);
	}
}

void *thread_User(void *Unused){
		char risposta[1024],string[1024],newString[1024];
		char *local;
		char posto[1024];
		int row,col,num;
		int i,c,r;
		while(1){
		scanf("%s",risposta);
```

```
if(strcmp(risposta,"status")==0){
        system("clear");
        riordinaFile();
        pthread_mutex_lock(&backup_mut);
        backup=fopen("server_backup_file.txt","r+");
        printf("Status del file backup: \n");
        while(fscanf(backup,"%s\n",string)!=EOF){
                local=strtok(string,"#");
                i=atoi(local);
                strcpy(newString,"");
                local=strtok(NULL,"#");
                while(local!=NULL){
                        num=atoi(local);
                        row=0;
                        col=0;
                        while(row*NUMCOL<num){
                                row++;
                        }
                        if(row*NUMCOL!=num)
                                row--;
                        col=num-row*NUMCOL;
                        sprintf(posto,"%d,%d",row,col);
                        strcat(newString,posto);
                        strcat(newString," ");
                        local=strtok(NULL,"#");
                }
                printf("CODE: %d VALUE: %s\n",i,newString);
        }
        fclose(backup);
        pthread_mutex_unlock(&backup_mut);
```

```
                printf("\nStatus attuale della sala: \n");
                pthread_mutex_lock(&mutex_a);
                for (r=0;r<NUMROW;r++){
                        printf("%d: ",r+1);
                        for(c=0;c<NUMCOL;c++){
                                if(matrix[r*NUMCOL +c]==0){
                                        printf("x ");
                                }
                                else{
                                        printf("%d ",r*NUMCOL+c);
                                }
                        }
                        printf("\n");
                }
                pthread_mutex_unlock(&mutex_a);
        }
        else{
                if(strcmp(risposta,"quit")==0){
                        printf("Good Bye\n");
                        exit(1);
                }
        }
        fflush(stdin);
}
}
int main(int argc, char **argv){
        if(argc>2){
                printf("Avvia il server senza parametri o specificando il port number\n");
        }
        int port_number;
```

```
if(argc==2){
	int charact=0;
	while(charact<strlen(argv[1])){
		if(argv[1][charact]<48||argv[1][charact]>57){
			printf("Input invalido un portNumber contiene solo numeri\n");
			exit(EXIT_FAILURE);
		}
		charact++;
	}
	port_number=atoi(argv[1]);
}
else{
	port_number=25001;
}
pthread_mutex_init(&init_mut,NULL);
pthread_mutex_init(&backup_mut,NULL);
sem_elem=semget(21,1,IPC_CREAT|0666);
semctl(sem_elem,0,SETVAL,0);
int on=1,r;
code=(char **)malloc(sizeof(char*)*NUMROW*NUMCOL);
for(r=0;r<NUMROW*NUMCOL;r++){
	code[r]=NULL;
}
initialize_server();
pthread_mutex_init(&mutex_a,NULL);
pthread_mutex_init(&mutex_d,NULL);
pthread_mutex_init(&mutex_socket,NULL);
pthread_mutex_init(&mutex_l,NULL);
pthread_mutex_init(&mutex_lp,NULL);
```

```
	struct sigaction act;
	sigset_t set;
	sigfillset(&set);
	sigprocmask(SIG_BLOCK,&set,NULL);
	act.sa_handler=pipe_handler;
	act.sa_mask=set;
	sigaction(SIGPIPE,&act,NULL);
	int listen_s,cons_s;
	listen_s=socket(AF_INET,SOCK_STREAM,0);
	setsockopt(listen_s,SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
	struct sockaddr_in addr;
	addr.sin_family=AF_INET;
	addr.sin_port=htons(port_number);
	addr.sin_addr.s_addr=INADDR_ANY;
	if(bind(listen_s,(struct sockaddr *)&addr,sizeof(addr))==-1){
		printf("Fatal error impossibile connettere il server si prega di riavviare
l'applicazione\n");
		exit(0);
	}
	else{
		printf("Server correttamente inizializzato\n");
	}
	pthread_t printThread;
	pthread_create(&printThread,NULL,thread_User,(void *)NULL);
	pthread_t backupThreadF;
	pthread_create(&backupThreadF,NULL,backupThread,(void *)NULL);
	listen(listen_s,50);
	struct sockaddr_in client_addr;
	pthread_t threads[NUMROW*NUMCOL];
	int client_size;
```

```
	int i=0;
	pthread_t sendert;
	pthread_create(&sendert,NULL,sender_thread,(void *)NULL);
	while(1){
		cons_s=accept(listen_s,(struct sockaddr*)&client_addr,&client_size);
		initialize_client(cons_s);
		pthread_create(threads+i,NULL,thread_f,(void *)&cons_s);
		i++;
	}
}
```

File headerS.h

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<signal.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<pthread.h>
#include<semaphore.h>
#include<sys/sem.h>
#include<sys/ipc.h>
#include <arpa/inet.h>
#include"describer.h"
#define fflush(stdin) while ((getchar())!='\n')
	typedef struct __cons_list{
	int sock;
	int available;
```

```
	struct __cons_list *next;
}cons_list;
pthread_mutex_t mutex_l;
pthread_mutex_t mutex_socket;
int sem_elem;
int matrix[NUMROW*NUMCOL];//voglio inizializzare tutto a 1
pthread_mutex_t mutex_a;
pthread_mutex_t mutex_lp;
pthread_mutex_t mutex_d;
pthread_mutex_t init_mut;
pthread_mutex_t backup_mut;
int actual_sock;
char **code;
int global_sock;
int sockets[NUMROW*NUMCOL];
int max_client=-1;
char postieffettivi[NUMROW*NUMCOL];
FILE *backup;
cons_list *header=NULL;
cons_list *tail=NULL;
typedef struct __message{
	int type; // 1=assegnazione posti, 2=disdire operazione, 3=gestione
	char *text;//contenuto del messaggio
	int sock_dest; //unused for type 1 and 2. Se type=3 e il messaggio lo invia il server allora sock_dest è il socket destinazione.
}msg;

typedef struct __list{
	msg *text;
	struct __list *next;
```

```
}list;
list *highprio_t=NULL;
list *lowprio_t=NULL;
list *highprio_h=NULL;
list *lowprio_h=NULL;

void removeSock(int sock);
int getCode(char *mex);
void riordinaFile();
void inserisciLista(list *lista, list *newElement,int num);
msg *ReadCommand(int sock);
int WriteCommand(msg* messaggio);
int ProcessCommand(msg *messaggio, int my_sock);
void removeSock(int sock){
        pthread_mutex_lock(&mutex_socket);
        int i;
        for(i=0;i<max_client+1;i++){
                if(sockets[i]==sock){
                        sockets[i]=-1;
                        close(sock);
                        if(i==max_client)
                                max_client--;

                }
        }
        pthread_mutex_unlock(&mutex_socket);
}
int getCode(char *mex){

        int i,count=0;
```

```
char *local_string;
char string[NUMROW*NUMCOL];
char *token;
for(i=0;i<NUMROW*NUMCOL;i++){
        if(code[i]==NULL){
                code[i]=(char *)malloc(sizeof(char)*strlen(mex)+ 6);
                if(code[i]==NULL){
                        printf("Impossibile fare malloc\n");
                }
                sprintf(code[i],"%s",mex);
                local_string=malloc(sizeof(char)*strlen(mex));
                strcpy(local_string,mex);
                token=strtok(local_string," ");
                sprintf(string,"%d",i);
                strcat(string,"#");
                strcat(string,token);
                token=strtok(NULL," ");
                while(token!=NULL){
                        strcat(string,"#");
                        strcat(string,token);
                        token=strtok(NULL," ");
                }
                pthread_mutex_lock(&backup_mut);
                backup=fopen("server_backup_file.txt","a");
                if(backup==NULL){
                                        printf("Fatal error\n");
                                        exit(0);
                                }
                if(fprintf(backup,"%s\n",string)<0){
                        printf("Fatal error\n");
```

```
                        }
                        fclose(backup);
                        pthread_mutex_unlock(&backup_mut);
                        return i;
                }
        }
        return -1;
}



void riordinaFile(){
        pthread_mutex_lock(&backup_mut);
        backup=fopen("server_backup_file.txt","r");
        int code;
        char c;
        char local_string[4096];
        char *local;
        char **codes;
        codes=(char**)malloc(sizeof(char*)*NUMROW*NUMCOL);
        char string[4096];
        while(fscanf(backup,"%s\n",string)!=EOF){
                strcpy(local_string,string);
                code=atoi(strtok(local_string,"#"));
                codes[code]=(char *)malloc(sizeof(char)*strlen(string));
                local=strtok(NULL,"#");
                if(local!=NULL){
                strcpy(codes[code],local);
                local=strtok(NULL,"#");
                while(local!=NULL){
```

```
                    strcat(codes[code],"#");
                    strcat(codes[code],local);
                    local=strtok(NULL,"#");
                }
                }else{
                    codes[code]=NULL;
                }
                }
        fclose(backup);
        backup=fopen("server_backup_file.txt","w");
        for(int i=0;i<NUMROW*NUMCOL;i++){
            if(codes[i]!=NULL){
                fprintf(backup,"%d#%s\n",i,codes[i]);
            }
        }
        fclose(backup);
        pthread_mutex_unlock(&backup_mut);
}
void inserisciLista(list *lista, list *newElement,int num){
        struct sembuf oper;
        oper.sem_num=0;
        oper.sem_op=1;
        newElement->next=NULL;
        if(num==1){
            pthread_mutex_lock(&mutex_l);
        }
        else{
            pthread_mutex_lock(&mutex_lp);
        }
        if(lista==NULL){
```

```
                                        if(num==1){
                                                highprio_h=newElement;
                                                highprio_t=newElement;
                                        }
                                        else{
                                                lowprio_h=newElement;
                                                lowprio_t=newElement;
                                        }
                                }
        else{
                        lista->next=newElement;
                        lista=newElement;
                }
        if(num==1){
                pthread_mutex_unlock(&mutex_l);
        }
        else{
                pthread_mutex_unlock(&mutex_lp);
        }
        semop(sem_elem,&oper,1);
}
```

msg *ReadCommand(int sock){ //La ReadCommand legge un byte per volta per assicurarsi di leggere uno e un solo messaggio alla volta. Evitiamo cosi lo scenario in cui piu messaggi arrivano alla socket e rimangono in attesa di lettura e viene letto più di uno.

```
        msg * messaggio=malloc(sizeof(msg));
        char character[3]=" ";
        int c=0;
        int x=2;
        int fine=0;
```

```
	char local[4096];
	ssize_t size;
		while(1){
leggi:		if(read(sock,character,x*sizeof(char))==0){
				return NULL;
			}
			if(x!=1){
			if(strcmp(character,"-a")==0){
				messaggio->type=1;
				x=1;
				character[0]=' ';
				character[1]='\0';
				goto leggi;
			}
			else{
			if(strcmp(character,"-d")==0){
				messaggio->type=2;
				x=1;
				character[0]=' ';
				character[1]='\0';
				goto leggi;
			}
			else{
			if(strcmp(character,"-c")==0){
				messaggio->type=3;
				x=1;
				goto leggi;
			}
			printf("Il messaggio ricevuto non ha senso connessione con client diverso da quello ufficiale... procedo alla disconnessione\n");
```

```
                removeSock(sock);
                }
                }
                }
                if(x==1){
                        if(strcmp(character,"-")==0){
                                fine=1;
                        }
                        else{
                                local[c]=character[0];
                                c++;
                        }
                }
                if(fine==1 && strcmp(character,"f")==0){
                        local[c-1]='\0';
                        messaggio->text=(char *)malloc(sizeof(char)*strlen(local));
                        strcpy(messaggio->text,local);
                        return messaggio;
                }
        }
}



int ProcessCommand(msg *messaggio, int my_sock){

        list *anotherElement=malloc(sizeof(list));
        int count=0;
        char string[NUMROW*NUMCOL];
```

```
int i=0;
int c=0;
int d=0;
char mess[1024];
int scritto=0;
char *local=(char*)malloc(sizeof(char)*50);
char *local_code=(char*)malloc(sizeof(char)*50);
int array[NUMROW*NUMCOL];
msg *risposta=malloc(sizeof(msg));
msg *risposta1=malloc(sizeof(msg));
list *newElement=malloc(sizeof(list));
pthread_mutex_lock(&mutex_a);
if(messaggio->type==1){
        strcpy(mess,messaggio->text);
        risposta->type=1;
        risposta->text=(char *)malloc(sizeof(char)*strlen(messaggio->text));
        strcpy(risposta->text,messaggio->text);
        local=strtok(messaggio->text," ");
        while(local!=NULL){
                array[d]=atoi(local);
                d++;
                local=strtok(NULL," ");
        }
        for(i=0;i<d;i++){
                if(matrix[array[i]]==0){
                        risposta->text="e";
                        risposta->type=3;
                        risposta->sock_dest=my_sock;
                        newElement->text=risposta;
                        inserisciLista(highprio_t,newElement,1);
```

```
                        pthread_mutex_unlock(&mutex_a);
                        return 0;
                }
        }
        for(i=0;i<d;i++){
                matrix[array[i]]=0;
        }
        risposta1->type=3;
        risposta1->sock_dest=my_sock;
        risposta1->text=malloc(sizeof(char)*10);
        sprintf(risposta1->text,"%d",getCode(mess));
        newElement->text=risposta1;
        inserisciLista(highprio_t,newElement,1);
        anotherElement->text=risposta;
        inserisciLista(lowprio_t,anotherElement,2);
}
if(messaggio->type==2){
        list *newElement=malloc(sizeof(list));
        local_code=strtok(messaggio->text," ");
        risposta->type=2;
        if(code[atoi(local_code)]!=NULL){
        risposta->text=(char*)malloc(sizeof(char)*strlen(code[atoi(local_code)]));
        strcpy(risposta->text,code[atoi(local_code)]);
        local=strtok(code[atoi(local_code)]," ");
        while(local!=NULL){
                matrix[atoi(local)]='1';
                local=strtok(NULL," ");
        }
        code[atoi(local_code)]=NULL;
        newElement->text=risposta;
```

```
inserisciLista(lowprio_t,newElement,2);
risposta1->text=malloc(sizeof(char)*4);
strcpy(risposta1->text," ct");
risposta1->type=3;
anotherElement->text=risposta1;
risposta1->sock_dest=my_sock;
if(my_sock==-1){
        free(anotherElement);
        return -1;
}
inserisciLista(lowprio_t,anotherElement,1);
pthread_mutex_lock(&backup_mut);
backup=fopen("server_backup_file.txt","a");
if(backup==NULL){
        printf("Fatal error\n");
        exit(0);
}
if(fprintf(backup,"%s\n",local_code)<0){
        printf("Fatal error\n");
        exit(0);
}
count=0;
fclose(backup);
pthread_mutex_unlock(&backup_mut);
}
else{
        risposta->type=3;
        risposta->sock_dest=my_sock;
        risposta->text=malloc(sizeof(char)*4);
        strcpy(risposta->text,"cnt");
```

```
                newElement->text=risposta;
                inserisciLista(highprio_t,newElement,1);
            }
        }
        pthread_mutex_unlock(&mutex_a);
}
        int WriteCommand(msg* messaggio){
            cons_list *local_head=header;                   //lista definita nel server come lista di
socket gestite
            pthread_mutex_lock(&mutex_socket);
            char testo[4096], buffer[4096];
            int done=0;
            int i;
            if(messaggio->type!=3){
                if(messaggio->type==1){
                    sprintf(testo,"-a %s -f",messaggio->text);
                }
                if(messaggio->type==2){
                    sprintf(testo,"-d %s -f",messaggio->text);
                }
                for(i=0;i<max_client+1;i++){
                    if(sockets[i]!=-1){
                        write(sockets[i],testo,strlen(testo));
                    }
                }
            }
            else{
                sprintf(buffer,"%s",messaggio->text);
                sprintf(testo,"-c %s -f",messaggio->text);
                actual_sock=messaggio->sock_dest;
```

```
                        if(write(actual_sock,testo,strlen(testo))==-1){
                                messaggio->type=2;
                                strcpy(messaggio->text,buffer);
                                ProcessCommand(messaggio,-1);
                        }
                        else{
                        done=1;
                        }
                }
                pthread_mutex_unlock(&mutex_socket);
                return done;
}
```

Codice Client

Anche il codice del client è strutturato similmente al codice del server

File client.c

```
#include "headerC.h"
#define fflush(stdin) while ((getchar())!='\n')
void alarm_handler(int sig){
        printf("Timeout scaduto closing connection\n");
        close(client_sock);
        printf("l'applicazione sta per chiudersi\n");
        exit(0);
}

void pipe_handler(int sig){
```

```
        printf("Lost connection with server\n");
        close(client_sock);
        printf("l'applicazione sta per chiudersi\n");
        exit(0);
}


void *polling_thread(void *i){
        sigset_t set;
        sigfillset(&set);
        sigdelset(&set,SIGPIPE);
        sigprocmask(SIG_BLOCK,&set,NULL);
        int sock=*(int*)i;
        msg *mex=(msg *)malloc(sizeof(msg));
        while(1){
                mex=ReadCommand(sock);
                if(mex==NULL){
                        printf("Server disconnected\n");
                        raise(SIGPIPE);
                        close(client_sock);
                        lost=1;
                }
                ProcessCommand(mex);
        }
}

int main(int argc, char **argv){
        if(argc!=3){
                printf("Errore devi inserire 2 valori di input\n ipNumber portNumber");
                exit(EXIT_FAILURE);
```

```
}
char *portNumber=argv[2];
char * ipNumber=argv[1];
printf("Welcome in the client\n");
posti=(char *)malloc(sizeof(char)*NUMROW*NUMCOL+1);
int result;
int c;
int on=1;
char* tipoLetto;
int contatore=0;
pthread_mutex_init(&av_mut,NULL);
pthread_mutex_lock(&av_mut);
pthread_mutex_init(&code_op,NULL);
pthread_mutex_lock(&code_op);
char st;
pthread_t thread;
struct sigaction act;
sigset_t set;
sigfillset(&set);
act.sa_handler=pipe_handler;
act.sa_mask=set;
sigaction(SIGPIPE,&act,NULL);
act.sa_handler=alarm_handler;
sigaction(SIGALRM,&act,NULL);
char *string=malloc(sizeof(char)*4096);
if(string==NULL){
        perror("Impossibile\n");
}
their_addr.sin_family=AF_INET;
their_addr.sin_port=htons(atoi(portNumber));
```

```
        inet_aton(ipNumber,&their_addr.sin_addr);
        char *converted;
        int type;
        pthread_create(&thread,NULL,polling_thread,(void *)&client_sock);
start:  while(1){
                if(lost){
                        alarm(0);
                        connection(their_addr,my_addr);
                        sleep(5);
                        stampaPosti();
                        lost=0;
                }
                printf("Scegliere l'operazione da eseguire: 0 per prenotare posti e 1 per disdire
prenotazione 2 per chiudere l'applicazione\n");
                alarm(300);

retry:          tipoLetto=malloc(sizeof(char)*1023);

                        if(fscanf(stdin,"%s",tipoLetto)!=EOF){
                        c=0;
                        while(c<strlen(tipoLetto)){

                                if(tipoLetto[c]<48||tipoLetto[c]>57){

                                        break;
                                }
                                c++;
                        }
                        if(c>=strlen(tipoLetto)-1)
                                type=atoi(tipoLetto);
```

```
                else{
                        printf("Inserisci un valore tra 0,1 o 2\n");
                        fflush(stdin);
                        free(tipoLetto);
                        goto retry;

                }
                free(tipoLetto);
        }
        fflush(stdin);
        alarm(500);
        if(type==0){
                stampaPosti();
        printf("Prego scegliere i posti da selezionare\n");
if(fgets(string,sizeof(char)*4096,stdin)==NULL ){
                goto start;
        }
        alarm(0);
        converted=convertiStringa(string);
        if(converted==NULL){
                printf("ERRORE RISPETTA LA SINTASSI\n");
        }
        else{
        WriteCommand(1,converted,client_sock);
        printf("Stiamo processando la tua richiesta rimani in attesa\n");
        pthread_mutex_lock(&av_mut);
        if(strcmp(code,"NULL")!=0)
                printf("Codice ricevuto: %s\n",code);
        printf("A breve sarà possibile effettuare una nuova operazione\n");
```

```
                sleep(10);
                stampaPosti();
                }
                }
                if(type==1){
                        stampaPosti();
                        printf("Inserire il codice di prenotazione\n");
reinserisci:    scanf("%s",string);
                                contatore=0;
                                while(contatore<strlen(string)){
                                        if(string[contatore]<48 || string[contatore]>57){
                                                printf("Ti ho chiesto di inserire un codice!\n");
                                                fflush(stdin);
                                                goto reinserisci;
                                        }
                                        contatore++;
                        }

                        alarm(0);
                        printf("Codice acquisito\n");
                        WriteCommand(2,string,client_sock);
                        pthread_mutex_lock(&code_op);
                        if(found!=0){
                        printf("Operazione conclusa good bye\n");
                        }
                        else{
                                printf("Impossibile completare la richiesta\n");
                        }
                        printf("A breve sarà possibile effettuare una nuova operazione\n");
                        sleep(10);
```

```
                stampaPosti();
            }
            if(type==2){
                printf("L'applicazione sta per chiudersi\n");
                sleep(2);
                exit(0);
            }
            if(type!=0&&type!=1&&type!=2){
                printf("Ti ho detto di inserire una cifra tra 0 1 e 2\n");
                goto start;
            }
        }
}
```

File headerC.h

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

```
int NUMROW,NUMCOL;
int found;
char *posti;
int client_sock;
int lost=1;
char code[2048];
pthread_mutex_t av_mut;
pthread_mutex_t code_op;
struct sockaddr_in their_addr;
struct sockaddr_in my_addr;
typedef struct __message{
        int type; // 1=assegnazione posti, 2=disdire operazione, 3=gestione
        char *text;//contenuto del messaggio
        int sock_dest; //unused for type 1 and 2. Se type=3 e il messaggio lo invia il server allora sock_dest è il socket destinazione.
}msg;
```

```
msg *ReadCommand(int sock){     //La ReadCommand legge un byte per volta per assicurarsi di leggere uno e un solo messaggio alla volta. Evitiamo cosi lo scenario in cui piu messaggi arrivano alla socket e rimangono in attesa di lettura e viene letto più di uno.
        msg * messaggio=malloc(sizeof(msg));
        char character[1]=" ";
        int c=0;
        int x=2;
        int fine=0;
        char local[4096];
        ssize_t size;
                if(read(sock,character,sizeof(char))==0){
                        return NULL;
```

```
}
if(character[0]=='-'){
        character[0]=' ';
        character[1]='\0';
        if(read(sock,character,sizeof(char))==0){
                return NULL;
        }
        switch(character[0]){
                case 'a':
                                messaggio->type=1;
                                break;
                case 'd':
                                messaggio->type=2;
                                break;
                case 'c':
                                messaggio->type=3;
                                break;
                default:
                                printf("Errore\n");
                                return NULL;
        }
        while(character[0]!='f'){
                if(read(sock,character,sizeof(char))==0){
                        return NULL;
                }
                if(character[0]=='-' || character[0]=='f'){
                }
                else{
                        local[c]=character[0];
                        c++;
```

```
                        }
                    }
                    local[c]='\0';
                }
        messaggio->text=malloc(sizeof(char)*strlen(local));
        strcpy(messaggio->text,local);
        return messaggio;
}


void WriteCommand(int type,char *string, int sock){
        char local[3];
        char *final_s;
        if(type==1){
                strcpy(local,"-a");
        }
        if(type==2){
                strcpy(local,"-d");
        }
        final_s=(char *)malloc(sizeof(char)*(strlen(string)+1+strlen(local)+3));
        sprintf(final_s,"%s %s -f",local,string);
        write(sock,final_s,strlen(final_s)*sizeof(char));
}

void stampaPosti(){
        system("clear");
        int r,c;
        for(r=0;r<NUMROW;r++){
                printf("%d  ",r);
                for(c=0;c<NUMCOL;c++){
```

```
                        if(posti[r*NUMCOL+c]=='0'){
                                printf("X");
                        }
                        else{
                                printf("%d",c);
                        }
                }
                printf("\n\n");
        }
}
void init(int sock){
        char stringa[4098];
        char *local,*col,*pos;
        alarm(30);
        if(read(sock,stringa,NUMROW*NUMCOL*sizeof(char)+50)==0){
                printf("Inizializzazione fallita a causa di una disconnessione del server\n");
                printf("Riavvio la procedura di connessione\n");
        }
        else{
                alarm(0);
                local=strtok(stringa,",");

                NUMROW=atoi(local);

                col=strtok(NULL,",");
                if(col==NULL){
                        printf("Errore riavvia l'applicazione\n");
                        exit(EXIT_FAILURE);
                }
```

```
                NUMCOL=atoi(col);
                pos=strtok(NULL,",");
                if(pos==NULL){
                        printf("Errore riavvia l'applicazione\n");
                        exit(EXIT_FAILURE);
                }
                posti=(char *)malloc(sizeof(char)*NUMROW*NUMCOL);
                strcpy(posti,pos);
                printf("Inizializzazione completata\n");
        }
}
void connection(struct sockaddr_in their_addr,struct sockaddr_in my_addr){
        int scelta;
        int on=1;
        client_sock=socket(AF_INET,SOCK_STREAM,0);
retry:  if(connect(client_sock,(struct sockaddr *)&their_addr,sizeof(their_addr))==-1){
                printf("Impossibile stabilire una connessione\n");
                printf("Inserire 0 per chiudere e 1 per riprovare\n");
                scanf("%d",&scelta);
                if(scelta==1){
                        sleep(2);
                        goto retry;
                }
                else{
                        exit(0);
                }
        }
        printf("Connessione con il server stabilita\n");
        init(client_sock);
}
```

```
char * convertiStringa(char *string){
	char *stringa=malloc(strlen(string)*sizeof(char));
	char **local=malloc(sizeof(char)*NUMROW*NUMCOL);
	char str_array[NUMROW*NUMCOL][1024];
	char *fila=malloc(sizeof(char)*10);
	char *poltrona=malloc(sizeof(char)*10);
	int c=0;
	int i=0;
	int n_poltrona,n_fila;
	local[i]=strtok(string," ");
	while(local[i]!=NULL){
		i++;
		local[i]=strtok(NULL," ");
	}
	while(c<i){
		fila=strtok(local[c],",");
		poltrona=strtok(NULL,",");
		if(fila==NULL || poltrona==NULL){
			return NULL;
		}
		n_fila=atoi(fila);
		n_poltrona=atoi(poltrona);
		if(n_poltrona>=NUMCOL || n_fila>=NUMROW || n_poltrona<0 || n_fila<0)
			return NULL;
		sprintf(str_array[c],"%d",n_poltrona+n_fila*NUMCOL);
		c++;
	}
```

```
        if(i==1){
                strcpy(stringa,str_array[0]);
        }
        else{
                strcat(str_array[0]," ");
                strcat(str_array[0],str_array[1]);
                strcpy(stringa,str_array[0]);
                c=2;
                while(c<i){
                        strcat(stringa," ");
                        strcat(stringa,str_array[c]);
                        c++;
                }
                strcat(stringa," ");
        }
        return stringa;
}

void ProcessCommand(msg *mex){
        char *string=malloc(sizeof(char)*50);
        int d=0, array[NUMROW*NUMCOL],i,valore;
        if(mex->type==1){
                string=strtok(mex->text," ");
                while(string!=NULL){
                        valore=atoi(string);
                        posti[valore]='0';
                        string=strtok(NULL," ");
                }
        }
        if(mex->type==2){
```

```
                string=strtok(mex->text," ");
                while(string!=NULL){
                        valore=atoi(string);
                        posti[valore]='1';
                        string=strtok(NULL," ");
                }
        }
        if(mex->type==3){
                string=strtok(mex->text," ");
                if(strcmp(string,"e")==0){
                        printf("Impossibile completare la richiesta perchè i posti sono già
occupati\n");
                        strcpy(code,"NULL");
                        pthread_mutex_unlock(&av_mut);
                }
                else{
                if(strcmp(string,"cnt")==0){
                        printf("Code not found\n");
                        found=0;
                        pthread_mutex_unlock(&code_op);
                }
                else{
                        if(strcmp(string,"ct")==0){
                                found=1;
                                pthread_mutex_unlock(&code_op);
                        }
                        else{
                                strcpy(code,mex->text);
                                pthread_mutex_unlock(&av_mut);
                        }
```

```
                }
        }
        free(mex);
}
        }
```

Infine ecco i codici per l'installer del server e i vari Makefile

File server_installer.c:

//Installer application for server

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
#include<string.h>
#include<sys/stat.h>
int main(){
                int newOut;
                printf("Welcome in the installer for server application\n");
                int row,col;
                char buffer[1024];
                printf("Please digit the number of rows and the number of place for rows in the follow format <numOfRows> <numOfPlaces>\n\n For example i have 6 rows and 5 places for rows so i digit 6 5 \n\n");
                scanf("%d %d",&row,&col);
                printf("Stiamo salvando le tue informazioni\n\n\n");
                int fd,dd,cd;
                fd=open("describer.h",O_CREAT|O_TRUNC|O_WRONLY,0666);
                sprintf(buffer,"\n#define NUMROW %d \n#define NUMCOL %d\n ",row,col);
                write(fd,buffer,strlen(buffer));
```

```
        int backup;
        backup=open("server_backup_file.txt",O_CREAT|O_TRUNC|O_WRONLY,0666);
        printf("Operazione completata\n");
}
```

Makefile server

```
install:
        gcc server_installer.c -o installer
        ./installer
        gcc server.c -o serverApplication -lpthread
clear:
        rm installer
        rm serverApplication
        rm server_backup_file.txt
        rm describer.h
```

Makefile client

```
install:
        gcc client.c -o client -lpthread
clear:
        rm client
```