Progetto di Sistemi Operativi, anno accademico 2015/2016

Realizzato da:

Cosimo Cinquilli 5777814 cosimo.cinquilli@stud.unifi.it
Matteo Moriani 5757129 matteo.moriani@stud.unifi.it

Data di consegna: 08/07/2016

INDICE:

1. ARCHITETTURA E SCELTE PROGETTUALI

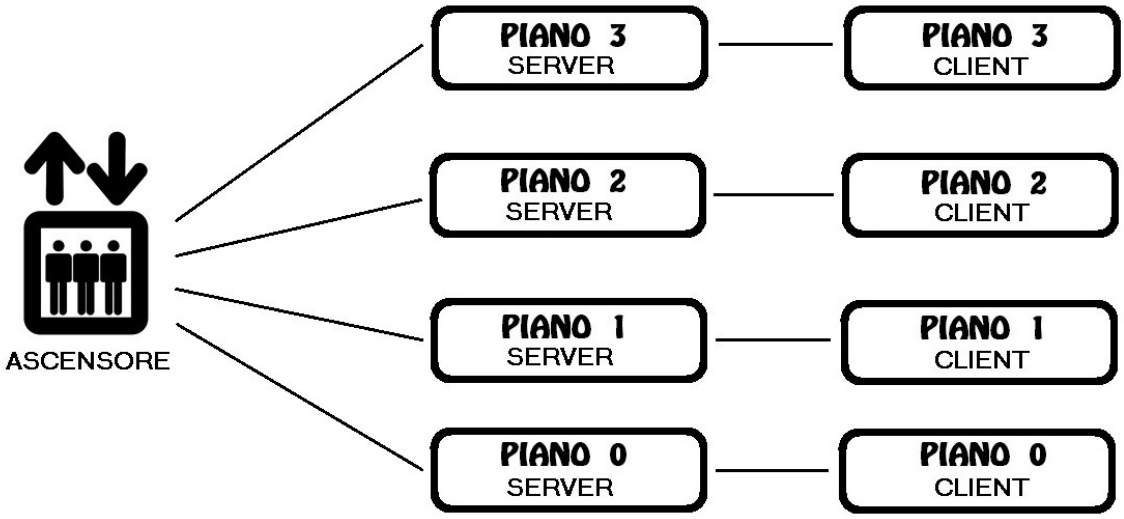

Illustrazione 1: schema dei server e dei client. Le linee di collegamento rappresentano connessioni su socket

Il progetto è strutturato in sette file. Quattro sono *file sorgente* C, due sono *header* e il rimanente è un *makefile*. Come si può vedere dalla illustrazione 1, abbiamo interpretato i piani come quattro server, ognuno dei quali si connette con il corrispettivo piano client, mentre l'ascensore è un client che si connette a tutti i piani server. Questa scelta risiede nella semplicità di programmazione e nella maggiore concorrenza che questo tipo di implementazione riesce a garantire.

La struttura dati più utilizzata è la lista, infatti ve ne sono cinque in totale: una per le persone all'interno dell'ascensore e una per le persone che attendono l'ascensore in ogni singolo piano.

2. ISTRUZIONI PER LA COMPILAZIONE E L'ESECUZIONE

1. Decomprimere l'archivio cinquillimoriani.tar.gz in una directory preferiblmente vuota

2. Spostarsi nella subdirectory “sources” e aggiungere i file di input dei quattro piani insieme ai file sorgente appena decompressi

3. Aprire una shell e spostarsi nella directory “sources” contenente i file sorgente e i file di input

4. Eseguire il comando “make all”

5. Aprire una seconda shell e spostarsi nella stessa directory

6. Digitare ./piani in una shell e ./ascensore nell'altra senza premere invio

7. Premere invio prima nella shell dove si è digitato ./piani e poi nell'altra in rapida successione per eseguire il programma con il limite temporale all’esecuzione di 5 minuti

8. Se si vuole eseguire il programma senza limite temporale, attendendo quindi il completo servizio di ogni persona generata, aggiungere al passo 6 al comando per lanciare i processi dei piani l’opzione “--fine-servizio” (senza virgolette)

3. CODICE CON SPIEGAZIONE DETTAGLIATA

“**persona.h**” definisce il tipo persona che ha come campi una stringa “nomeTipo” (Adulto, Bambino o Addetto alla consegna), un intero per il peso e un altro intero per la destinazione, ovvero quale piano vuole raggiungere la persona. Sempre qui vengono definite staticamente le tre stringhe che servono nella stampe all'interno dei file di log. Viene infine definito il prototipo della funzione creaPersona che verrà implementata da persona.c.

```
#ifndef PERSONA_H_
#define PERSONA_H_

typedef struct _persona {
	char* nomeTipo;
	int peso;
	int destinazione;
} Persona;

static char *nomiTipi[3]={"Adulto","Bambino","Addetto alla consegna"};

Persona creaPersona(char tipo, int destinazione);

#endif
```

“**linkedlist.h**” oltre a definire un modo più comodo per riferirsi al tipo nodo_lista_persone, definisce le funzioni che poi verranno implementate da linkedlist.c. Inoltre crea il tipo lista_persone che possiede un puntatore head (che punta alla testa della lista) e un puntatore current (che punta alla fine della lista). Include persona.h poiché necessita il tipo di dati Persona nelle dichiarazioni. Elenca i prototipi delle funzioni implementate in linkedList.c.

```
#include "persona.h"
#ifndef LINKEDLIST_H_
#define LINKEDLIST_H_
typedef struct _nodo_lista_persone nodo_lista_persone;
typedef struct _nodo_lista_persone
{
	Persona* persona;
	nodo_lista_persone* next;
}nodo_lista_persone;

typedef struct _lista_persone
{
	nodo_lista_persone* head;
	nodo_lista_persone* current;
} lista_persone;
lista_persone* crea_lista_persone();
nodo_lista_persone* getHead(lista_persone* lista);
nodo_lista_persone* aggiungi_alla_lista(lista_persone* lista, Persona*
persona_da_aggiungere);
nodo_lista_persone* cerca_per_tipo(lista_persone* lista, char *tipo,
nodo_lista_persone **prev);
int cancella_per_tipo(lista_persone* lista, char *tipo);
Persona* cancella_scesa(lista_persone* lista, int arrivo);
#endif
```

“**persona.c**” include “persona.h” e possiede un'unica funzione creaPersona che riceve in ingresso il carattere codiceTipo (A,B o C: il tipo di persona che sale sull'ascensore) e l'intero destinazione (il piano che la persona deve raggiungere) e restituisce nuova_persona. Alla nuova persona creata verranno passati il tipo e la destinazione date in ingresso, inoltre in base al tipo le verrà assegnato attraverso uno switch un peso diverso (80 per A, 40 per B, 90 per C) e un nome tipo diverso(Adulto, Bambino, Addetto alle consegne).

```
#include "persona.h"

Persona creaPersona(char tipo, int destinazione) {
	Persona nuova_persona;
	switch (tipo){
	case 'A':
		nuova_persona.peso = 80;
		nuova_persona.nomeTipo=nomiTipi[0];
		break;
	case 'B':
		nuova_persona.peso = 40;
		nuova_persona.nomeTipo=nomiTipi[1];
		break;
	case 'C':
		nuova_persona.peso = 90;
		nuova_persona.nomeTipo=nomiTipi[2];
		break;
	}
	nuova_persona.destinazione=destinazione;
	return nuova_persona;
}
```

“**linkedList.c**” include “linkedList.h” più altre librerie di utilità per la scrittura di messaggi di errore, per l'allocazione della memoria e per la gestione delle stringhe. Come prima cosa la funzione crea_lista_persone alloca la memoria per contenere la lista, inizializza a NULL i suoi valori head e current e restituisce il puntatore alla lista.

La funzione getHead serve a restituire rapidamente il valore a cui punta head, ovvero un nodo di nodo_lista_persone, ogniqualvolta ve ne fosse bisogno.

La funzione crea_testa riceve in ingresso il puntatore alla lista, quello a persona_da_aggiungere e restituisce un nuovo puntatore di un nodo appena aggiunto alla testa della lista. Questo puntatore punta inizialmente al primo bit di uno spazio in RAM creato con la funzione malloc. E' prevista anche la gestione del fallimento della creazione del puntatore, dovuto per esempio alla RAM piena e quindi impossibilitata a fornire lo spazio necessario. Viene poi assegnato al campo “persona” della struttura a cui punta “nodo” il puntatore persona_da_aggiungere, mentre a “next” NULL. Questa funzione infine assegna ai puntatori nodo, current e head lo stesso indirizzo.

aggiungi_alla_lista aggiunge il puntatore persona_da_aggiungere alla testa della lista in caso ancora non ce ne sia una invocando crea_testa, altrimenti alloca

in memoria dello spazio per il nodo ed il puntatore relativo. Nel caso in cui il puntatore non punti a niente è prevista inoltre la stampa dell'errore. La persona a cui punta nodo diventa ora persona_da_aggiungere, next di nodo non punta a niente mentre next di current punta ora a nodo, poi current punta a nodo, cioè viene costantemente aggiornato in modo che punti all'ultimo nodo creato che infine viene restituito.

cerca_per_tipo restituisce il primo nodo persona a partire dalla testa del tipo dato in ingresso, mentre ritorna NULL se non esiste. Scrive inoltre in prev il puntatore al nodo precedente al quello eventualmente trovato.

cancella_per_tipo elimina il primo nodo di un certo tipo cercato attraverso cerca_per_tipo, aggiorna i puntatori della lista e libera la memoria allocata al nodo cancellato.

cancella_scesa cancella il primo nodo in lista che contiene il puntatore a persona con la destinazione uguale a quella data in ingresso (trovato con *cerca_per_destinazione*) e ritorna il puntatore alla persona contenuta nel nodo appena eliminato.

cerca_per_destinazione restituisce il primo nodo che contiene in puntatore alla persona con la stessa destinazione passata come parametro, se esiste. Infine aggiorna prev con il puntatore al nodo precedente a quello trovato.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "linkedList.h"

lista_persone* crea_lista_persone(){
	lista_persone* lista = (lista_persone*)malloc(sizeof(lista_persone));
	lista->head=NULL;
	lista->current=NULL;
	return lista;
}

nodo_lista_persone* getHead(lista_persone* lista) {
	return lista->head;
}

nodo_lista_persone* crea_testa(lista_persone* lista, Persona*
persona_da_aggiungere) {
	nodo_lista_persone* nodo = (nodo_lista_persone*)
malloc(sizeof(nodo_lista_persone));
	if (NULL == nodo) {
		printf("\n Creazione del nodo della linked list fallita!\n");
		return NULL;
	}
	nodo->persona = persona_da_aggiungere;
	nodo->next = NULL;

	lista->head = lista->current = nodo;

	return nodo;
}

nodo_lista_persone* aggiungi_alla_lista(lista_persone* lista, Persona*
persona_da_aggiungere) {
```

```
	if (NULL == lista->head) {
		return (crea_testa(lista, persona_da_aggiungere));
	}
	nodo_lista_persone *nodo = (nodo_lista_persone*)
malloc(sizeof(nodo_lista_persone));
	if (NULL == nodo) {
		printf("\n Creazione del nodo della linked list fallita! \n");
		return NULL;
	}

	nodo->persona=persona_da_aggiungere;

	nodo->next = NULL;

	lista->current->next = nodo;
	lista->current = nodo;
	return nodo;
}

nodo_lista_persone* cerca_per_tipo(lista_persone* lista, char *tipo,
nodo_lista_persone **prev) {
	nodo_lista_persone *nodo = lista->head;
	nodo_lista_persone *tmp = NULL;

	int found = 0;

	while (nodo != NULL) {
		if (strcmp(nodo->persona->nomeTipo, tipo) == 0) {
			found = 1;
			break;
		} else {
			tmp = nodo;
			nodo = nodo->next;
		}
	}
	if (found) {
		if (prev) {
			*prev = tmp;
		}
		return nodo;
	} else {
		return NULL;
	}
}

int cancella_per_tipo(lista_persone* lista, char *tipo) {
	nodo_lista_persone* prev = NULL;
	nodo_lista_persone *del = NULL;


	del = cerca_per_tipo(lista,tipo,&prev);

	if(del){
		if (prev != NULL) {
				prev->next = del->next;
			}
			if (del == lista->current) {
				lista->current = prev;
			}
			if (del == lista->head) {
```

```
                              lista->head = del->next;
                        }
                        free(del->persona);
                        free(del);
        } else {
                return -1;
        }
}


nodo_lista_persone* cerca_per_destinazione(lista_persone* lista, int
destination, nodo_lista_persone** prev) {
        nodo_lista_persone *nodo = lista->head;
        nodo_lista_persone *tmp = NULL;
        int found = 0;

        while (nodo != NULL) {
                if (nodo->persona->destinazione == destination) {
                        found = 1;
                        break;
                } else {
                        tmp = nodo;
                        nodo = nodo->next;
                }
        }

        if (found) {
                        if (prev) {
                                *prev = tmp;
                        }
                        return nodo;
                } else {
                        return NULL;
                }
}
Persona* cancella_scesa(lista_persone* lista, int arrivo) {
        nodo_lista_persone *prev = NULL;
        nodo_lista_persone *del = NULL;
        Persona* cancellata = NULL;

        del = cerca_per_destinazione(lista, arrivo, &prev);

        if (del) {
                if (prev != NULL) {
                        prev->next = del->next;
                }
                if (del == lista->current) {
                        lista->current = prev;
                }
                if (del == lista->head) {
                        lista->head =del->next;

                }
                cancellata = del->persona;
                free(del);
                return cancellata;
        }
        return cancellata;
}
```

“**piani.c**” come prima cosa include persona.h e linkedlist.h, definisce il valore zero ogniqualvolta verrà usato DEFAULT_PROTOCOL e dichiara due interi costanti statici: 1 per indicare al piano server la connessione del piano_client, 0 per quella dell'ascensore. Dopodiché vengono dichiarati tre array di stringhe contenenti i nomi dei file da usare per istanziare i socket, i nomi dei file di input e i nomi dei file di log. Vengono inoltre dichiarati l'intero numero_piano, la variabile di tempo “tempo_avvio” che serve a tener traccia del tempo di sistema al fine di consentire la sincronizzazione all'avvio dei server e dei client e la variabile di tempo “tempo_terminazione” che inizializzato al momento dell'avvio costituisce il tempo limite per l'esecuzione del programma.

Le procedure *invia_nel_socket* e ricevi_*dal_socket* servono a semplificare l’invio e la ricezione di dati attraverso il socket gestendo eventuali errori. Fanno uso nel loro corpo delle system call read e write per attuare la comunicazione.

La procedura “*client*” come prima cosa apre nome_file_input del relativo piano in sola lettura, con gestione dell'errore in caso il contenuto di inputFp (input file pointer) sia NULL. Dopodiché client esegue ciclicamente le seguenti istruzioni: legge la linea corrente per poi spostarsi a quella successiva (che verrà letta nel ciclo seguente) finché non verranno lette tutte le righe (se è appena stata letta l'ultima termina); prova a connettersi con il socket a lui associato. Se fallisce termina generando un errore, mentre se ha successo salva le tre caratteristiche della persona della riga (convertendo in interi dove necessario). Dopodiché genera la persona con queste caratteristiche e si ferma finché non sarà giunto il tempo di inserirla fra le persone in coda. Dopo aver comunicato al server che vuole connettersi, gli invierà la persona. Infine chiude il socket e ricomincia il ciclo. Finito il ciclo, chiude il file di input e termina.

La procedura “*server*”, in modo analogo a client apre il file di log nome_file_log relativo al piano, in sola scrittura, dopodiché dichiara gli interi e le struct necessari ad istanziare un singolo socket prima del ciclo di accept. L'istruzione unlink elimina il socket se già esiste, bind lo crea e listen crea una coda massima di due connessioni. Il ciclo di accept prevede: l'accettazione della connessione del socket (che attende finché non ce n'è almeno una in coda), che restituisce il descrittore di file che verrà usato per comunicare con ascensore o piano client. Se a connettersi è il piano client, sul puntatore “nuovo_arrivo” viene salvato l'indirizzo del primo bit dello spazio allocato dalla funzione malloc nel quale verrà salvata la persona appunto nuova arrivata. Successivamente si controlla che ci sia un nuovo arrivo, vengono lette le informazioni relative alle sue caratteristiche, viene aggiunto alla lista e infine vengono registrate nel file di log tali informazioni più il tempo di generazione.

Nel caso si connetta invece l'ascensore, prima del suo ciclo while inizializza il peso con la lettura del peso rimanente disponibile, effettuata tramite il socket clientFd.

All'interno del while invece fintanto che c'è una testa della lista d'attesa sul piano (quindi finché c'è almeno una persona) e finché l'ascensore può contenerne altre, viene cancellata una persona dalla coda d'attesa, aggiornato il peso disponibile dell’ascensore e viene inviata all'ascensore la persona. Uscita dal

ciclo di accept, la procedura controlla che sia passato il tempo limite d'esecuzione per poi, in caso, terminare.

La procedura *leggi_parametri* in fase di lancio del programma discerne se avviarlo con o senza il limite temporale. Infine il main costituisce la funzione che lancia tutti i processi tramite le system call fork che a partire dal singolo processo “piani” si duplica due volte per creare i quattro piani, che a loro volta si duplicano per differenziarsi in client e server. Inoltre fa in modo che vengano lanciati in concomitanza sincronizzando il tempo di avvio e che terminino attendendo ognuno la terminazione del proprio figlio tramite waitpid.

```
#include <unistd.h> /* write, lseek, close, exit */
#include <sys/stat.h> /*open */
#include <fcntl.h> /*open*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>   /*  Per socket AF_UNIX */
#include <time.h>
#include <string.h>
#include <errno.h>

#include "persona.h"
#include "linkedList.h"

#define DEFAULT_PROTOCOL 0
#define DURATA_MAX_MINUTI 5

static const int CONNESSIONE_PIANO_CLIENT = 1;
static const int CONNESSIONE_ASCENSORE = 0;

static const char* NOME_SOCKET_PIANO[4] = { "piano0.sock", "piano1.sock",
            "piano2.sock", "piano3.sock" };
static const char* NOME_FILE_INPUT[4] =
            { "piano0", "piano1", "piano2", "piano3" };
static const char* NOME_FILE_LOG[4] = { "piano0.log", "piano1.log",
            "piano2.log", "piano3.log" };
time_t tempo_avvio;
time_t tempo_terminazione;
int numero_piano;

enum terminazione {
      cinque_minuti, fine_servizio
} terminazione = cinque_minuti;

void invia_nel_socket(int SocketFd, const void* buffer, size_t dim) {
      int scritto = write(SocketFd, buffer, dim);
      if (scritto < dim) {
            char* s;
            asprintf(&s, "Errore invio sul socket \"%s\", terminazione piano
                  %i...\n", NOME_SOCKET_PIANO[numero_piano], numero_piano);
            perror(s);
            exit(36);
      }
}

void ricevi_dal_socket(int SocketFd, void* nuovo_arrivo, size_t dim) {
      int letto = read(SocketFd, nuovo_arrivo, dim);
      if (letto < 0) {
```

```
			char* s;
			asprintf(&s, "Errore ricezione dal socket \"%s\", terminazione piano
					%i...\n",NOME_SOCKET_PIANO[numero_piano], numero_piano);
			exit(36);
		}
}

void client() {
		char* tipo = NULL;
		char* tempo_generazione = NULL;
		char* destinazione = NULL;
		int tempo_generazione_num = 0;
		int destinazione_num = 0;
		FILE * inputFp = NULL;

		printf("Eseguo client piano%i\n", numero_piano);

		inputFp = fopen(NOME_FILE_INPUT[numero_piano], "r");

		if (inputFp == NULL) {
			char* s;
			asprintf(&s, "Impossibile aprire file di input \"%s\", terminazione
					client piano %i ",NOME_FILE_INPUT[numero_piano],
					numero_piano);
			perror(s);
			exit(-3);
		}
		long int posizione = 0;

		while (1) {
			char* riga = NULL;
			size_t len = 0;
			int presente = 0;
			int tempo;

			//legge una riga dal file di input e genera la persona
			//se la riga e' vuota, termina
			getline(&riga, &len, inputFp);

			if (strcmp(riga, "\n") == 0) {
				printf("Raggiunta fine del file di input \"%s\", terminazione
						client piano %i\n",
						NOME_FILE_INPUT[numero_piano], numero_piano);
				break;
			}

			int SocketFd;
			int SocketLenght;
			struct sockaddr_un SocketAddress;
			struct sockaddr* SocketAddrPtr;
			SocketAddrPtr = (struct sockaddr*) &SocketAddress;
			SocketLenght = sizeof(SocketAddress);

			/* Create a UNIX socket, bidirectional, default protocol */
			SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
			if (SocketFd == -1) {
				char* s;
				asprintf(&s, "Errore nella creazione del socket \"%s\",
						terminazione client piano %i ",
						NOME_SOCKET_PIANO[numero_piano], numero_piano);
```

```
                perror(s);
                exit(76);
            }

            SocketAddress.sun_family = AF_UNIX; /* Set domain type */
            strcpy(SocketAddress.sun_path, NOME_SOCKET_PIANO[numero_piano]); /*
                        Set name */

            int connesso = connect(SocketFd, SocketAddrPtr, SocketLenght);
            if (connesso == -1) {
                char* s;
                asprintf(&s, "Client piano %i NON CONNESSO", numero_piano);
                perror(s);
                fclose(inputFp);
                exit(35);
            }
            char* rigaTmp = riga;

            tipo = strsep(&rigaTmp, " ");
            tempo_generazione = strsep(&rigaTmp, " ");
            destinazione = strsep(&rigaTmp, " ");

            tempo_generazione_num = atoi(tempo_generazione);
            destinazione_num = atoi(destinazione);

            Persona persona = creaPersona(tipo[0], destinazione_num);
            free(riga);
            tempo = time(NULL);
            sleep(tempo_generazione_num - (tempo - tempo_avvio));

            invia_nel_socket(SocketFd,
                &CONNESSIONE_PIANO_CLIENT, sizeof(CONNESSIONE_PIANO_CLIENT));

            long unsigned dimensione = sizeof(persona);
            //invia la persona
            invia_nel_socket(SocketFd, &persona, dimensione);

            //invia la lunghezza della stringa
            dimensione = strlen(persona.nomeTipo);
            invia_nel_socket(SocketFd, &dimensione, sizeof(dimensione));

            //invia la stringa
            invia_nel_socket(SocketFd, persona.nomeTipo, dimensione);

            close(SocketFd);
            if (tempo_terminazione <= time(NULL)) {
                printf("Raggiunto limite temporale, terminazione client piano
                            %i\n",
                            numero_piano);
                break;
            }
        }

        fclose(inputFp);
}

void server() {
        lista_persone* coda = NULL;
        nodo_lista_persone* testa = NULL;
        coda = crea_lista_persone();
```

```
FILE* logFp = NULL;
Persona* nuovo_arrivo = NULL;
int connessione = -1;
time_t ora;

logFp = fopen(NOME_FILE_LOG[numero_piano], "w");

printf("Eseguo server piano%i\n", numero_piano);
if (logFp < 0) {
      char* s;
      asprintf(&s, "Impossibile aprire file di log \"%s\", terminazione
                  server piano %i...",
                  NOME_FILE_LOG[numero_piano], numero_piano);
      perror(s);
      exit(-3);
}
fprintf(logFp, "Avvio del sistema: %s (%i)\n", ctime(&tempo_avvio),
            (int)tempo_avvio);

unlink(NOME_SOCKET_PIANO[numero_piano]);
int SocketFd;
int SocketLenght;
struct sockaddr_un SocketAddress;
struct sockaddr* SocketAddrPtr;

SocketAddrPtr = (struct sockaddr*) &SocketAddress;
SocketLenght = sizeof(SocketAddress);

/* Create a UNIX socket, bidirectional, default protocol */
SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
if (SocketFd == -1) {
      printf("Errore nella creazione del socket \"%s\", termnazione
                  server piano %i...",
                  NOME_SOCKET_PIANO[numero_piano], numero_piano);
      exit(-76);
}
SocketAddress.sun_family = AF_UNIX; /* Set domain type */
strcpy(SocketAddress.sun_path, NOME_SOCKET_PIANO[numero_piano]); /* Set
           name */
int result = bind(SocketFd, SocketAddrPtr, SocketLenght);
if (SocketFd == -1) {
      printf("Errore nella bind del socket \"%s\", termnazione server
                  piano %i...",
                  NOME_SOCKET_PIANO[numero_piano], numero_piano);
      perror("");
      exit(-76);
}
result = listen(SocketFd, 2);
if (SocketFd == -1) {
      printf("Errore nella listen del socket \"%s\", termnazione
                  server piano %i...",
                  NOME_SOCKET_PIANO[numero_piano], numero_piano);
      perror("");
      exit(-76);
}

while (1) {
      int clientFd = accept(SocketFd, SocketAddrPtr, &SocketLenght);
      if(clientFd==-1){
```

```
		perror("Errore nella connessione con i client");
		printf("Terminazione server piano %i",numero_piano);
		exit(5);
	}
	//ricevi_dal_socket(clientFd, &connessione, sizeof(connessione));
	read(clientFd, &connessione, sizeof(connessione));

	if (connessione == CONNESSIONE_PIANO_CLIENT) {// si e' connesso un
		piano-client
		printf("Server piano %i, connessione piano client\n",
			numero_piano);

		nuovo_arrivo = (Persona*) malloc(sizeof(Persona));
		if(nuovo_arrivo==NULL){
			printf("Errore allocazione memoria per ricezione
				persona, terminazione server piano %i...",
				numero_piano);
			exit(37);
		}

		ricevi_dal_socket(clientFd, nuovo_arrivo, sizeof(Persona));

		//legge lunghezza striga nomeTipo
		long unsigned dimensione = 0;
		ricevi_dal_socket(clientFd, &dimensione, sizeof(dimensione));

		//legge stringa nomeTipo
		nuovo_arrivo->nomeTipo = (char*) malloc(dimensione);
		ricevi_dal_socket(clientFd, nuovo_arrivo->nomeTipo,
				dimensione);

		aggiungi_alla_lista(coda, nuovo_arrivo);

		ora = time( NULL);
		int tempo_generazione = ora - tempo_avvio;

		printf("[GENERATO] %s al piano %i, destinazione piano %i,
				tempo dall'avvio %i, %s\n",
				nuovo_arrivo->nomeTipo, numero_piano,
				nuovo_arrivo->destinazione, tempo_generazione,
				ctime(&ora));
		fprintf(logFp,"[GENERATO] %s, destinazione piano %i, tempo
				dall'avvio %i, %s\n",
				nuovo_arrivo->nomeTipo, nuovo_arrivo-
				>destinazione, tempo_generazione,
				ctime(&ora));

	} else if (connessione == CONNESSIONE_ASCENSORE) {// si e' connesso
		l'ascensore
		printf("Server piano %i, connessione ascensore\n",
				numero_piano);
		int peso = 0;
		int presente = 0;
		//riceve il peso massimo caricabile dall'ascensore
		ricevi_dal_socket(clientFd, &peso, sizeof(int));
		while (1) {
			testa = getHead(coda);
			if (testa == NULL) {
				presente = 0;
				invia_nel_socket(clientFd, &presente,
```

```
                                    sizeof(int));
                              close(clientFd);
                              break;
                        }
                        peso = peso - testa->persona->peso;
                        if (peso < 0) {
                              presente = 0;
                              invia_nel_socket(clientFd, &presente,
                                    sizeof(int));
                              close(clientFd);
                              break;
                        }

                        //comunica all'ascensore che ci sono persone da inviare
                        presente = 1;
                        invia_nel_socket(clientFd, &presente, sizeof(int));
                        //write(clientFd, &presente, sizeof(int));

                        //invia la persona
                        invia_nel_socket(clientFd, testa->persona,
                                    sizeof(Persona));
                        //write(clientFd, testa->persona, sizeof(Persona));

                        //invia la dimensione della stringa nomeTipo
                        long unsigned dimensione = strlen(testa→persona-
                                    >nomeTipo);

                        invia_nel_socket(clientFd, &dimensione,
                                          sizeof(dimensione));

                        //invia la stringa nomeTipo
                        invia_nel_socket(clientFd, testa->persona->nomeTipo,
                                          dimensione);

                        cancella_per_tipo(coda, testa->persona->nomeTipo);
                  }
                  close(clientFd);
            } else {
                  printf("Errore di connessione");
                  continue;
            }
            if (tempo_terminazione <= time(NULL)) {
                  printf("Raggiunto limite temporale. ");
                  break;
            }
      }
      close(SocketFd);
      printf("Terminazione server piano%i\n", numero_piano);
      ora = time(NULL);
      fprintf(logFp,"Terminazione piano, %s (%i)\n", ctime(&ora), (int)ora);
      fclose(logFp);
}

void leggi_parametri(int argc, char* argv[]) {
      if(argc==1){
            printf("Esecuzione con limite temporale di 5 minuti\n");
            return;
      }
      if (argc == 2 && strcmp(argv[1], "--fine-servizio") == 0) {
            terminazione = fine_servizio;
```

```
            printf("Esecuzione senza limite temporale, fino alla completa
            lettura dei file di input e servizio di ogni passegero\n");
      } else {
            printf(
                  "Uso: %s [--fine-servizio]\nIl programma termina di default
                  dopo 5 minuti di esecuzione.\n"
                  "--fine-servizio: il programma continua fino alla completa
                  lettura dei file di input e servizio di tutti i passeggeri\n",
                  argv[0]);
            exit(1);
      }
}

int main(int argc, char * argv[]) {
      int status = 0;
      numero_piano = 0;
      int pidserver1 = 0;
      int pidserver2 = 0;
      int pidserver3 = 0;

      leggi_parametri(argc, argv);

      tempo_avvio = time(NULL) + 3;
      int prima_fork = 0;
      int pid = fork();
      if (pid) {
            prima_fork++;
      }
      pidserver2=pid;

      pid = fork();
      //assegna i numeri dei piani differenziandoli in base ai risultati delle
      fork
      //e salva anche i pid dei figli (che saranno i piani server) per la
      terminazione
      if (pid) {
            if (prima_fork) {
                  numero_piano = 2;
                  pidserver3=pid;
            } else {
                  numero_piano = 0;
                  pidserver1=pid;
            }
      } else {
            if (prima_fork) {
                  numero_piano = 3;
            } else {
                  numero_piano = 1;
            }
      }

      pid = fork();
      time_t now = time(NULL);
      tempo_terminazione = now + (DURATA_MAX_MINUTI * 60);

      if (!pid) {
            sleep(tempo_avvio - now + 2);
            tempo_avvio = time(NULL);
            client();
      } else {
```

```
            sleep(tempo_avvio - now);
            tempo_avvio = time(NULL) + 2;
            server();
            //aspetta terminazione del piano client corrispondente
            waitpid(pid, &status,0);
      }
      //per server piano 2 risulta pidserver3 != 0 e aspetta che server piano 3
            termini
      if(pidserver3) waitpid(pidserver3,&status,0);
      //per server piano 0 risulta pidserver1 != 0 e aspetta che server piano 1
            termini
      if(pidserver1) waitpid(pidserver1, &status, 0);
      //per server piano 0 risulta pidserver2 != 0 e aspetta che server piano 2
            termini
      if(pidserver2) waitpid(pidserver2,&status,0);
      return status;
}
```

“**ascensore.c**” come prima cosa include persona.h e linkedlist.h, definisce il valore zero ogniqualvolta verrà usato DEFAULT_PROTOCOL, 3 per TEMPO_SOSTA, 3 per TEMPO_SPOSTAMENTO e 300 per il PESO_MASSIMO sopportato dall'ascensore. Vengono poi dichiarati gli enumeratori per le direzioni, i piani e le terminazioni, la costante statica che indica la connessione dell'ascensore e l'array di stringhe con i nomi dei socket dei piani. Infine tre variabili (num_adulti, num_bambini e num_addetti) usate per tenere il conto di quante persone di ciascun tipo vengono servite dall’ascensore al fine di redigere il riassunto dell’attività giornaliera.

La procedura *muovi_ascensore* sposta l'ascensore di un piano verso l'alto o verso il basso in base alla direzione corrente.

Anche in questo sorgente sono presenti le funzioni *invia_nel_socket* e *ricevi_nel_socket*, implementate in modo analogo a piani.c.

La funzione *carica_persone* riceve in ingresso il descrittore del socket sul quale comunicare e il puntatore al file di log sul quale registrare l'arrivo delle persone, mentre restituisce il numero di persone caricate sull'ascensore. Legge se c'è almeno una persona in attesa: se non ce n'è restituisce 0, mentre se c'è entra in un ciclo while. Fintanto che c'è una persona in attesa, la copia in nuovo arrivo che a sua volta verrà aggiunto alla lista delle persone sull'ascensore. Infine stampa tutte le informazioni relative alle persone salite sull'ascensore, le salva nel file di log e aggiorna peso complessivo e quantità delle persone a bordo.

La procedura *scarica_persone* riceve in ingresso il file di log e si comporta in modo analogo a carica_persone, con la differenza che non ha bisogno di operare sul socket e che all'inizio del ciclo elimina la persona indicata come “scesa” e libera lo spazio in memoria da essa precedentemente occupato. Inoltre, per ogni persona, aumenta il contatore delle persone servite di un particolare tipo (ovvero incrementa una variabile tra num_bambini, num_adulti e num_addetti, scelta a seconda del tipo della persona scesa).

Infine la funzione *main* dopo tutte le dichiarazioni necessarie crea la lista delle persone sull'ascensore (inizialmente vuota), apre il file di log, e poi

fintanto che l'ascensore non è ancora partito per la prima volta (quindi è ancora fermo al piano 0 e non ci sono persone) ciclicamente istanzierà il socket con il quale comunicherà con il piano server 0 e tenterà di connettersi. Se non ci riesce termina, altrimenti comunica al server il codice di riconoscimento dell'ascensore, calcola il peso massimo che può imbarcare l'ascensore e lo comunica al server. Successivamente richiama carica_persone, registra quante ne ha caricate, chiude il socket e aspetta 10 ms. Se in questo ciclo ha caricato almeno una persona, ne esce. Uscito da questo ciclo entra in quello principale: richiama muovi_ascensore, attende 3 secondi (il tempo di spostamento da un piano all'altro) e poi altri 3 (il tempo di sosta al piano). Quindi richiama scarica_persone, istanzia nuovamente il socket per provare a comunicare con il piano server. Se fallisce registra nell'apposito array che il piano è terminato, cambiando il valore da zero a uno e diminuisce di uno l'intero che conta il numero di piani attivi. Se in corrispondenza del piano c'è già il valore 1, controlla che l'intero piani_attivi sia a 0, in quel caso termina anche l'ascensore. Se invece riesce a connettersi scrive sul socket il codice di riconoscimento dell'ascensore, calcola e invia il peso massimo imbarcabile e poi richiama carica_persone. Infine chiude il socket e, fuori dal ciclo, viene chiuso il file di log.

```c
#include <sys/socket.h>
#include <sys/un.h>   /*   Per socket AF_UNIX */
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#include "persona.h"
#include "linkedList.h"

#define PESO_MASSIMO 300
#define DEFAULT_PROTOCOL 0
#define TEMPO_SOSTA 3
#define TEMPO_SPOSTAMENTO 3

enum direzione {
      ALTO, BASSO
} direzione;
enum piano {
      piano0, piano1, piano2, piano3
} piano;

enum terminazione {
      cinque_minuti, fine_file
} terminazione;

static const int CONNESSIONE_ASCENSORE = 0;

static const char* NOME_SOCKET_PIANO[4] = { "piano0.sock", "piano1.sock",
            "piano2.sock", "piano3.sock" };

lista_persone* lista = NULL;
int carico = 0;
time_t tempo_avvio;
int num_bambini = 0;
int num_adulti = 0;
```

```
int num_addetti = 0;

void muovi_ascensore() {
	if (direzione == ALTO) {
		if (piano == piano3) {
			direzione = BASSO;
			piano--;
		} else {
			piano++;
		}
	} else {
		if (piano == piano0) {
			direzione = ALTO;
			piano++;
		} else {
			piano--;
		}
	}
}

void invia_nel_socket(int SocketFd, const void* buffer, size_t dim) {
	int scritto = write(SocketFd, buffer, dim);
	if (scritto < dim) {
		char* s;
		asprintf(&s,"Errore invio al piano %i, terminazione ascensore ",
			piano);
		perror(s);
		exit(36);
	}
}

void ricevi_dal_socket(int SocketFd, void* nuovo_arrivo, size_t dim) {
	int letto = read(SocketFd, nuovo_arrivo, dim);
	if (letto < 0) {
		char* s;
		asprintf(&s, "Errore ricezione dal piano %i, terminazione ascensore
			",piano);
		perror(s);
		exit(36);
	}
}

int carica_persone(int SocketFd, FILE* logFp) {
	int presente = 0;

	ricevi_dal_socket(SocketFd, &presente, sizeof(int));
	if (!presente) {
		return 0;
	}
	Persona* nuovo_arrivo = (Persona*) malloc(sizeof(Persona));
	int numero_caricate = 0;
	while (presente) {
		nuovo_arrivo = (Persona*) malloc(sizeof(Persona));
		//printf("ricezione persona");
		ricevi_dal_socket(SocketFd, nuovo_arrivo, sizeof(Persona));
		aggiungi_alla_lista(lista, nuovo_arrivo);

		//legge dimensione stringa nomeTipo
		long unsigned dimensione = 0;
		//printf("ricezione dimensione");
```

```
            ricevi_dal_socket(SocketFd, &dimensione, sizeof(dimensione));
            nuovo_arrivo->nomeTipo = (char*) malloc(dimensione);
            //legge la stringa nome tipo
            //printf("ricezione tipo");
            ricevi_dal_socket(SocketFd, nuovo_arrivo->nomeTipo, dimensione);

            int tempo_generazione = time(NULL) - tempo_avvio;
            time_t ora = time( NULL);
            printf(
                        "[SALITO] %s al piano %i, destinazione %i, tempo
                        dall'avvio %i, %s\n",
                        nuovo_arrivo->nomeTipo, piano, nuovo_arrivo-
                        >destinazione,
                        tempo_generazione, ctime(&ora));
            fprintf(logFp,
                        "[SALITO] %s al piano %i, destinazione %i, tempo
                        dall'avvio %i, %s\n",
                        nuovo_arrivo->nomeTipo, piano, nuovo_arrivo
                        ->destinazione,
                        tempo_generazione, ctime(&ora));

            carico = carico + nuovo_arrivo->peso;
            numero_caricate++;
            ricevi_dal_socket(SocketFd, &presente, sizeof(int));
      }
      return numero_caricate;
}

void scarica_persone(FILE* logFp) {
      Persona* scesa = NULL;
      scesa = cancella_scesa(lista, piano);
      while (scesa != NULL) {
            int tempo_scesa = time(NULL) - tempo_avvio;
            time_t ora = time( NULL);
            printf("[SCESO] %s al piano %i, tempo dall'avvio %i, %s\n",
                        scesa->nomeTipo, piano, tempo_scesa, ctime(&ora));
            fprintf(logFp, "[SCESO] %s al piano %i, tempo dall'avvio %i, %s\n",
                        scesa->nomeTipo, piano, tempo_scesa, ctime(&ora));
            switch (scesa->peso){
                  case 80:
                        num_adulti++;
                        break;
                  case 40:
                        num_bambini++;
                        break;
                  case 90:
                        num_addetti++;
                        break;
            }
            carico = carico - scesa->peso;
            free(scesa);
            scesa = cancella_scesa(lista, piano);
      }
}

int main(int argc, char *argv[]) {
      int peso_massimo_imbarcabile = PESO_MASSIMO;
      piano = 0;
      short piani_terminati[4] = { 0, 0, 0, 0 };
      int piani_attivi = 4;
```

```
lista = crea_lista_persone();
sleep(4);
tempo_avvio = time(NULL);

FILE* logFp = NULL;
logFp = fopen("ascensore.log", "w");
fprintf(logFp, "Avvio del sistema: %s (%i)\n", ctime(&tempo_avvio),
            (int) tempo_avvio);
int persone_caricate = 0;
do {
      int SocketFd, SocketLenght, tempo;
      struct sockaddr_un SocketAddress;
      struct sockaddr* SocketAddrPtr;

      SocketAddrPtr = (struct sockaddr*) &SocketAddress;
      SocketLenght = sizeof(SocketAddress);

      /* Create a UNIX socket, bidirectional, default protocol */
      SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
      SocketAddress.sun_family = AF_UNIX; /* Set domain type */
      strcpy(SocketAddress.sun_path, NOME_SOCKET_PIANO[piano]);
      int connesso = connect(SocketFd, SocketAddrPtr, SocketLenght);
      if (connesso == -1) {
            char* s;
            asprintf(&s,"Ascensore NON CONNESSO tramite socket \"%s\"\n",
                  NOME_SOCKET_PIANO[piano]);
            perror(s);
            close(SocketFd);
            exit(34);
      }

      invia_nel_socket(SocketFd, &CONNESSIONE_ASCENSORE,
            sizeof(CONNESSIONE_ASCENSORE));

      peso_massimo_imbarcabile = PESO_MASSIMO - carico;
      invia_nel_socket(SocketFd, &peso_massimo_imbarcabile, sizeof(int));

      persone_caricate = carica_persone(SocketFd, logFp);
      close(SocketFd);
      usleep(10000);
} while (persone_caricate == 0);

printf("Peso ascensore = %i\n", carico);

while (42) {
      muovi_ascensore();
      sleep(TEMPO_SPOSTAMENTO);
      printf("Arrivo al piano %i\n", piano);
      sleep(TEMPO_SOSTA);
      scarica_persone(logFp);
      int SocketFd, SocketLenght, tempo;
      struct sockaddr_un SocketAddress;
      struct sockaddr* SocketAddrPtr;

      SocketAddrPtr = (struct sockaddr*) &SocketAddress;
      SocketLenght = sizeof(SocketAddress);

      /* Create a UNIX socket, bidirectional, default protocol */
      SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
```

```
            SocketAddress.sun_family = AF_UNIX; /* Set domain type */
            strcpy(SocketAddress.sun_path, NOME_SOCKET_PIANO[piano]);
            int connesso = connect(SocketFd, SocketAddrPtr, SocketLenght);

            if (connesso != 0) {
                  if (piani_terminati[piano]) {
                        if (piani_attivi == 0) {
                              close(SocketFd);
                              break;
                        }
                        continue;
                  } else {
                        piani_attivi--;
                        piani_terminati[piano] = 1;
                        continue;
                  }
            } else {
                  printf("Connessione tramite \"%s\"\n",
                        NOME_SOCKET_PIANO[piano]);
            }

            invia_nel_socket(SocketFd, &CONNESSIONE_ASCENSORE,
                  sizeof(CONNESSIONE_ASCENSORE));

            peso_massimo_imbarcabile = PESO_MASSIMO - carico;
            invia_nel_socket(SocketFd, &peso_massimo_imbarcabile, sizeof(int));

            persone_caricate = carica_persone(SocketFd, logFp);
            printf("Peso ascensore = %i\n", carico);
            close(SocketFd);
      }
      printf("Terminazione ascensore...\n");
      time_t tempo_terminazione = time(NULL);
      fprintf(logFp, "Terminazione ascensore %s\n", ctime(&tempo_terminazione));
      char* s;
      asprintf(&s, "Riassunto attivita' giornaliera:\nTotale persone trasportate
                  %i\nBambini %i\nAdulti %i\nAddetti alle consegne %i\n",
                  num_bambini + num_adulti + num_addetti, num_bambini,
                  num_adulti, num_addetti);
      printf("%s", s);
      fprintf(logFp, "%s", s);
      fclose(logFp);
      return 0;
}
```

Il **makefile,** oltre ai target necessari alla compilazione del progetto (degno di nota il target **all** che avvia la compilazione di tutto il progetto), include alcuni target che possono risultare utili per la rimozione dei file creati dai processi durante l'esecuzione, nello specifico: il target **cleansocket** elimina i file ".sock" creati per la comunicazione tramite socket, il target **cleanlog** elimina i file ".log" generati dai processi per registrare le loro attività, il target **clean** elimina gli eseguibili e i codici oggetto generati dalla compilazione del progetto stesso. Il target **cleanall** esegue tutte le operazioni di cancellazione insieme.

4. SCREENSHOT E COMMENTO DEI FILE DI LOG

```
Avvio ascensore: Wed Jul  6 11:06:02 2016
[SALITO] Bambino al piano 0, destinazione 3, tempo dall'avvio 2, Wed Jul  6 11:06:04 2016
[SALITO] Adulto al piano 1, destinazione 0, tempo dall'avvio 11, Wed Jul  6 11:06:13 2016
[SALITO] Bambino al piano 1, destinazione 0, tempo dall'avvio 11, Wed Jul  6 11:06:13 2016
[SALITO] Adulto al piano 1, destinazione 2, tempo dall'avvio 11, Wed Jul  6 11:06:13 2016
[SCESO] Adulto al piano 2, tempo dall'avvio 17, Wed Jul  6 11:06:19 2016
```

Illustrazione 2: Estratto dalla parte iniziale del log dell'ascensore

Come si può vedere dall'illustrazione 2 il processo ascensore scrive nel file di log l'orario di avvio. Inizia quindi il servizio al piano 0 e non appena carica la prima persona parte verso il piano 1. Notare come in ogni riga sia annotato se sia salito o sceso un passeggero, seguito da tutte le informazioni dettagliate su: il tipo di passeggero, la destinazione, il tempo in secondi dall'avvio del programma e l'orario in cui avviene la salita o discesa.

```
[SCESO] Bambino al piano 2, tempo dall'avvio 340, Wed Jul  6 11:11:42 2016
[SCESO] Bambino al piano 3, tempo dall'avvio 346, Wed Jul  6 11:11:48 2016
Terminazione ascensore Wed Jul  6 11:12:12 2016
Riassunto attivita' giornaliera:
Totale persone trasportate 72
Bambini 30
Adulti 26
Addetti alle consegne 16
```

Illustrazione 3: Estratto dalla parte finale del log dell'ascensore

Nell'illustrazione 3 si vede come a fine del file di log il processo ascensore scriva l'orario di terminazione seguito dal riassunto dell'attività giornaliera, dove viene riportato il conteggio totale dei passeggeri trasportati e quello delle singole tipologie.

```
Avvio del piano: Wed Jul  6 11:06:02 2016
[GENERATO] Bambino, destinazione piano 3, tempo dall'avvio 2, Wed Jul  6 11:06:04 2016
[GENERATO] Adulto, destinazione piano 2, tempo dall'avvio 16, Wed Jul  6 11:06:18 2016
[GENERATO] Addetto alla consegna, destinazione piano 1, tempo dall'avvio 24, Wed Jul  6 11:06:26 2016
[GENERATO] Bambino, destinazione piano 3, tempo dall'avvio 28, Wed Jul  6 11:06:30 2016
[GENERATO] Addetto alla consegna, destinazione piano 2, tempo dall'avvio 35, Wed Jul  6 11:06:37 2016
```

Illustrazione 4:Estratto della parte iniziale del log del piano 0

Infine l'illustrazione 4 mostra come nel file di log di un piano (in questo caso il piano 0) sia riportato ancora l'orario di avvio del processo e tutte le

informazioni delle persone generate su quel piano, ovvero: tipo, destinazione, tempo in secondi dall’avvio del processo e orario di generazione.