

---

# Progetto di Sistemi Operativi

Anno accademico 2015/2016

---

Realizzato da:

- Daniel Zanchi
- Daniele Landi
- Federico Guerri

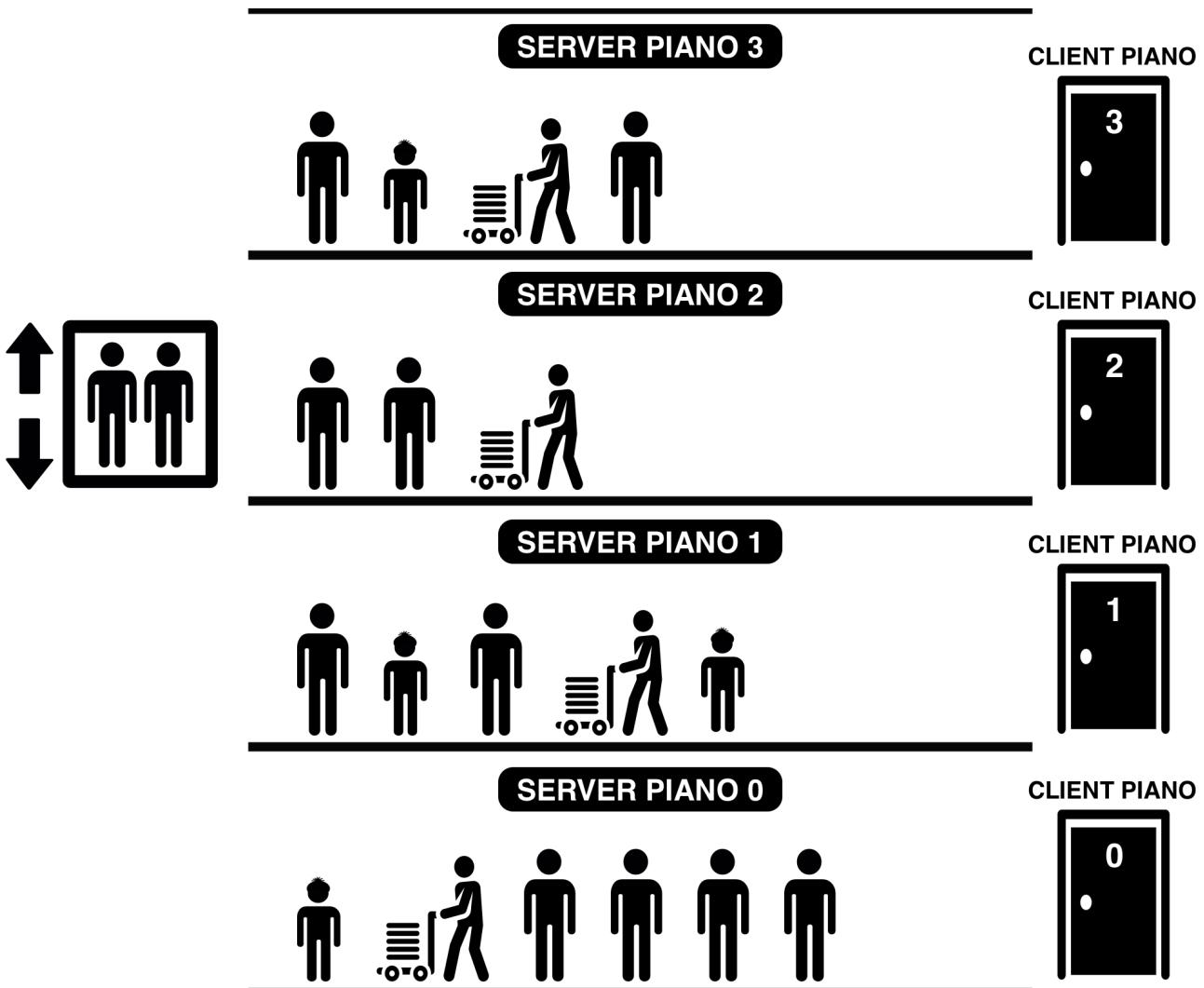


---

## ISTRUZIONI

1. Decomprimere il file guerrilandizanchi.zip
2. Posizionarsi nella sottocartella “src” ed aggiungere i file relativi ai quattro piani
3. Lanciare una Shell ed eseguire il comando “make all” (assicurarsi di essere posizionati all’interno della cartella “src”)
4. Digitare il comando “./piani” nella Shell
5. Aprire una seconda Shell e digitare il comando “./ascensore” (assicurandosi di star operando nella directory “src”)
6.
  - a) Eseguire quindi, in rapida successione, i due comandi (prima “./piani” poi “./ascensore”). In questo modo sarà eseguito il programma con un limite temporale di 5 minuti
  - b) Alternativamente, sostituire il comando al punto 4 con “./piani -termina” al fine di terminare il programma solo al completamento del servizio per ogni persona generata

## SCHEMA PROGETTO



Per realizzare il progetto abbiamo scelto di adottare la soluzione raffigurata dallo schema in figura. Ogni piano si compone di un server, che ospita più persone generate dal rispettivo piano client, quest'ultime vengono memorizzate in delle liste. Infine l'ascensore è un client che si connette, via via, ai piani server ed utilizza anch'esso una lista per tener traccia delle persone al suo interno.

---

## DESCRIZIONE FILES

“**persona.h**” viene definito il tipo persona, che ha tre campi:

- “categoriaPersona (Adulto, Bambino o Addetto alla consegna) stringa che indica la categoria alla quale appartiene ogni individuo;
- “peso” un intero per registrare il peso;
- “destinazione” un intero per memorizzare la destinazione;

Vengono inoltre definite le tre stringhe contenenti le informazioni relative alle varie categorie (che verranno registrate nel file di Log).

Infine si definisce il prototipo della funzione *creaPersona* che verrà poi implementata nel file persona.c.

```
1 #ifndef PERSONA_H_
2 #define PERSONA_H_
3
4 typedef struct _persona {
5     char* categoriaPersona;
6     int peso;
7     int destinazione;
8 } Persona;
9
10 static char *categoriePersone[3] = { "Adulto", "Bambino", "Addetto alla consegna" };
11
12 Persona creaPersona(char tipo, int destinazione);
13
14 #endif /* PERSONA_H_ */
```

---

“**persona.c**” include al suo interno l’header “**persona.h**” ed utilizza una funzione *creaPersona* che riceve in ingresso il carattere inerente alla categoria (varia in base alla persona generata) e l’intero “destinazione” (il piano di destinazione relativo alla persona). Restituisce, quindi, “nuova\_persona”.

I campi di quest’ultima saranno quindi inizializzati, rispettivamente con:

- La destinazione;
- La categoria alla quale la “nuova\_persona” appartiene;
- Il peso;

```
1 #include "persona.h"
2
3 Persona creaPersona(char categoria, int destinazione) {
4     Persona nuova_persona;
5     nuova_persona.destinazione = destinazione;
6     switch (categoria) {
7         case 'A':
8             nuova_persona.categoriaPersona = categoriePersone[0];
9             nuova_persona.peso = 80;
10            break;
11        case 'B':
12            nuova_persona.categoriaPersona = categoriePersone[1];
13            nuova_persona.peso = 40;
14            break;
15        case 'C':
16            nuova_persona.categoriaPersona = categoriePersone[2];
17            nuova_persona.peso = 90;
18            break;
19    }
20    return nuova_persona;
21 }
```

---

“**linkedlist.h**” viene utilizzato per far riferimento al tipo “nodo\_lista\_persone” in modo più semplice. Inoltre definisce le funzioni che verranno implementate nel file “**linkedlist.c**”.

Crea il tipo “lista\_persone” che possiede un puntatore “testaLista” (punta alla testa della lista) e un puntatore “elementoCorrente” (punta alla fine della lista).

Include “**persona.h**” dato che utilizza il tipo di dati “**Persona**”.

```
1 #include "persona.h"
2 #ifndef LINKEDLIST_H_
3 #define LINKEDLIST_H_
4
5 typedef struct _nodo_lista_persone nodo_lista_persone;
6
7 typedef struct _nodo_lista_persone {
8     Persona* persona;
9     nodo_lista_persone* successivo;
10 } nodo_lista_persone;
11
12 typedef struct _lista_persone {
13     nodo_lista_persone* testaLista;
14     nodo_lista_persone* elementoCorrente;
15 } lista_persone;
16
17 lista_persone* creaListaPersone();
18
19 nodo_lista_persone* getTestaLista(lista_persone* lista);
20
21 nodo_lista_persone* aggiungiPersonaLista(lista_persone* lista, Persona* persona_da_aggiungere);
22
23 nodo_lista_persone* ricercaPerTipo(lista_persone* lista, char *tipo, nodo_lista_persone **precedente);
24
25 void eliminaPerTipo(lista_persone* lista, char *tipo);
26
27 Persona* eliminaDiscesa(lista_persone* lista, int arrivo);
28
29 #endif /* LINKEDLIST_H_ */
```

---

“**linkedList.c**” include “**linkedList.h**” oltre ad alcune librerie necessarie per messaggi di errore, l’allocazione della memoria e per la gestione delle stringhe.

*creaListaPersone* viene allocata la memoria necessaria per la lista, quindi inizializza “*testaLista*” e “*elementoCorrente*” con “Null” e restituisce il puntatore alla lista.

*getTestaLista* restituisce il valore puntato da “*testaLista*”.

*creaTesta* tramite il puntatore alla lista e quello a “*persona\_da\_aggiungere*” restituisce un puntatore (ad uno spazio in Ram appena creato) di un nodo appena aggiunto in testa alla lista. Viene poi assegnato al campo “*persona*” della struttura a cui punta “*nodo*” il puntatore “*persona\_da\_aggiungere*”, mentre a “*successivo*” il valore “Null”. Infine si assegna ai puntatori “*testaLista*”, “*elementoCorrente*” e “*nodo*” lo stesso indirizzo.

*aggiungiPersonaLista* aggiunge il puntatore “*persona\_da\_aggiungere*” alla testa della lista (invocando, se necessario, *creaTesta*), altrimenti provvede ad allocare dello spazio per il nodo ed il puntatore. A questo punto, viene costantemente aggiornato il puntatore di “*nodo*”, in modo che punti all’ultimo nodo creato che verrà infine restituito.

*ricercaPerTipo* restituisce il primo nodo a partire dalla testa della lista relativo alla categoria data in ingresso, restituisce “Null” altrimenti. Viene inoltre salvato in “*precedente*” il puntatore al nodo precedente a quello restituito.

*eliminaPerTipo* elimina, aggiorna i puntatori e libera la memoria, il primo nodo appartenente ad una certa categoria (tramite *ricercaPerTipo*).

*eliminaDiscesa* elimina il primo nodo, nella lista, contenente il puntatore a “*persona*” con destinazione uguale a quella fornita come parametro (utilizza *ricercaPerDestinazione*) e restituisce il puntatore alla persona contenuta nel nodo eliminato.

*ricercaPerDestinazione* ritorna il primo nodo, della lista, che contiene un puntatore alla “*persona*” con la stessa destinazione passata come parametro. Poi aggiorna “*precedente*” con il puntatore al nodo precedente a quello trovato.

```

1 #include <stdlib.h>
2 #include <string.h>
3 #include "linkedList.h"
4
5 v lista_persone* creaListaPersone() { // Viene inizializzata la lista che conterrà le persone
6     lista_persone* lista = (lista_persone*) malloc(sizeof(lista_persone));
7     lista->testaLista = NULL;
8     lista->elementoCorrente = NULL;
9     return lista;
10 }
11
12 v nodo_lista_persone* getTestaLista(lista_persone* lista) { // Restituisce la testa della lista
13     return lista->testaLista;
14 }
15
16 v nodo_lista_persone* creaTesta(lista_persone* lista, Persona* persona_da_aggiungere) { // Aggiunge alla lista di
17     persone la nuova persona, gestendo anche l'eventuale fallimento
18     nodo_lista_persone* nodo = (nodo_lista_persone*) malloc(sizeof(nodo_lista_persone));
19     if (nodo == NULL) {
20         printf("\n Creazione del nodo della linked list fallita!\n");
21         return NULL;
22     }
23     nodo->persona = persona_da_aggiungere;
24     nodo->successivo = NULL;
25
26     lista->testaLista = lista->elementoCorrente = nodo;
27
28     return nodo;
29 }
30
31 v nodo_lista_persone* aggiungiPersonaLista(lista_persone* lista, Persona* persona_da_aggiungere) { //Crea una lista se
32     non c'è già, altrimenti alloca in memoria spazio per nodo e puntatore
33     if (lista->testaLista == NULL) {
34         return (creaTesta(lista, persona_da_aggiungere));
35     }
36     nodo_lista_persone *nodo = (nodo_lista_persone*) malloc(sizeof(nodo_lista_persone));
37     if (nodo == NULL) {
38         printf("\n Creazione del nodo della linked list fallita! \n");
39         return NULL;
40     }
41
42     nodo->persona = persona_da_aggiungere;
43     nodo->successivo = NULL;
44     lista->elementoCorrente->successivo = nodo;
45     lista->elementoCorrente = nodo;
46
47     return nodo;
48 }
49
50 v nodo_lista_persone* ricercaPerTipo(lista_persone* lista, char *tipo, nodo_lista_persone **precedente) { //Restituisce
51     il primo puntatore relativo al tipo di persona trovata, salvando il precedente nodo in precedente
52     nodo_lista_persone *nodo_temp = NULL;
53     nodo_lista_persone *nodo = lista->testaLista;
54     int found = 0;
55
56     while (nodo != NULL) {
57         if (strcmp(nodo->persona->categoriaPersona, tipo) == 0) {
58             found = 1;
59             break;
60         } else {
61             nodo_temp = nodo;
62             nodo = nodo->successivo;
63         }
64     }
65     if (found) {
66         if (precedente) {
67             *precedente = nodo_temp;
68         }
69     }
70 }
71
72 v void stampaLista(lista_persone* lista) {
73     nodo_lista_persone *nodo = lista->testaLista;
74
75     while (nodo != NULL) {
76         printf("Nome: %s Cognome: %s\n", nodo->persona->nome, nodo->persona->cognome);
77         nodo = nodo->successivo;
78     }
79 }
80
81 v void liberaLista(lista_persone* lista) {
82     nodo_lista_persone *nodo = lista->testaLista;
83
84     while (nodo != NULL) {
85         nodo_temp = nodo;
86         nodo = nodo->successivo;
87         free(nodo_temp);
88     }
89 }
90
91 v void stampaPersonale(Persona* persona) {
92     printf("Nome: %s Cognome: %s\n", persona->nome, persona->cognome);
93     printf("Eta: %d\n", persona->eta);
94     printf("Città: %s\n", persona->citta);
95     printf("Città: %s\n", persona->provincia);
96     printf("Città: %s\n", persona->pais);
97 }
98
99 v void liberaPersonale(Persona* persona) {
100    free(persona);
101 }

```

---

```

66     return nodo;
67 } else {
68     return NULL;
69 }
70 }
71
72 void eliminaPerTipo(lista_persone* lista, char *tipo) { //Cancella del tutto la prima persona del tipo specificato,
73     liberando anche la memoria
74     nodo_lista_persone * eliminata = NULL;
75     nodo_lista_persone* precedente = NULL;
76
77     eliminata = ricercaPerTipo(lista, tipo, &precedente);
78
79     if (eliminata) {
80         if (precedente != NULL) {
81             precedente->successivo = eliminata->successivo;
82         }
83         if (eliminata == lista->elementoCorrente) {
84             lista->elementoCorrente = precedente;
85         }
86         if (eliminata == lista->testaLista) {
87             lista->testaLista = eliminata->successivo;
88         }
89         free(eliminata->persona);
90         free(eliminata);
91     } else {
92         return ;
93     }
94 }
95 nodo_lista_persone* ricercaPerDestinazione(lista_persone* lista, int destination, nodo_lista_persone** precedente) {
96     //Restituisce la prima persona trovata con destinazione uguale a "destination", aggiorna anche "precedente"
97     nodo_lista_persone *nodo = lista->testaLista;
98     nodo_lista_persone *nodo_temp = NULL;
99     int found = 0;
100
101    while (nodo != NULL) {
102        if (nodo->persona->destinazione == destination) {
103            found = 1;
104            break;
105        } else {
106            nodo_temp = nodo;
107            nodo = nodo->successivo;
108        }
109    }
110
111    if (found) {
112        if (precedente) {
113            *precedente = nodo_temp;
114        }
115        return nodo;
116    } else {
117        return NULL;
118    }
119 }
120 Persona* eliminaDiscesa(lista_persone* lista, int arrivo) { //Cancella la prima occorrenza che ha destinazione uguale
121     ad "arrivo", restituisce il puntatore al nodo appena eliminato
122     nodo_lista_persone *precedente = NULL;
123     nodo_lista_persone * eliminata = NULL;
124     Persona* cancellata = NULL;
125
126     eliminata = ricercaPerDestinazione(lista, arrivo, &precedente);
127
128     if (eliminata) {
129         if (precedente != NULL) {
130             precedente->successivo = eliminata->successivo;
131         }

```

---

---

```
131     if (eliminata == lista->elementoCorrente) {
132         lista->elementoCorrente = precedente;
133     }
134     if (eliminata == lista->testaLista) {
135         lista->testaLista = eliminata->successivo;
136     }
137     cancellata = eliminata->persona;
138     free(eliminata);
139     return cancellata;
140 }
141 return cancellata;
142 }
143 }
```

---

“**piani.c**” include “**persona.h**” e “**linkedlist.h**”, definisce “**DEFAULT\_PROTOCOL**” oltre alle costanti per differenziare le connessioni al piano server : “1” per quella del “**piano\_client**”, “0” per l’ascensore. Vengono dichiarati tre array per identificare: i socket; i file di input; i file di Log. Si dichiarano infine: l’intero “**numero\_piano**”; la variabile “**tempo\_avvio**” che permette la sincronizzazione all’avvio dei server e dei client; la variabile “**tempo\_terminazione**” che fissa il limite temporale per l’esecuzione del programma.

*scriviNelSocket* e *leggiDalSocket* permettono la comunicazione tramite i socket tramite le system call “**read**” e “**write**”.

*client* innanzi tutto apre “**piani\_file\_input**” del relativo piano in sola lettura, con gestione dell’errore in caso il contenuto di “**input\_file\_piani**” (input file pointer) sia “Null”. Dopodiché *client* esegue, ciclicamente, le seguenti istruzioni:

1. Legge in successione tutte le righe del file (una ad ogni ciclo), termina una volta raggiunta la fine;
2. Tenta la connessione al socket associato. Se ha successo salva le caratteristiche della persona inerente alla riga appena letta.
3. Genera la persona, che sarà aggiunta alle persone in coda solo in un secondo momento.
4. Invia al server la persona appena generata.
5. Chiude il socket e ricomincia il ciclo. Finito il quale termina.

*server*, come il client, apre il file di Log “**files\_log**” relativo al piano, dopodiché dichiara interi e structs necessari alla creazione di un socket . Questo viene fatto prima del ciclo relativo all’istruzione “**accept**”. Quest’ultimo prevede:

1. L’accettazione della connessione del socket (che attende finché non ce n’è almeno una), che, permette, tramite il descrittore di file restituito di comunicare con l’ascensore o il piano client.
2.
  - a) Se a connettersi è il piano client, sul puntatore “**nuovo\_arrivato**” viene salvato l’indirizzo del primo bit dello spazio allocato tramite “**malloc**”, nel quale verrà salvata la persona appunto nuova arrivata. A questo punto, si controlla che si verifichi un nuovo arrivo, si leggono le informazioni relative alle caratteristiche e si aggiunge alla lista. Infine si registrano quest’ultime nel file di Log, insieme al tempo di generazione.
  - b) Nel caso si connetta l’ascensore: prima del suo ciclo “**while**” si inizializza il peso tramite la lettura (usando il socket “**clientFd**”) del carico rimanente. All’interno del

---

“while” invece, finché la lista relativa al piano ha una testa, ossia è presente almeno una persona, e finché l’ascensore è in grado di ospitarne altre: viene cancellata una persona dalla coda d’attesa; aggiornato il peso disponibile dell’ascensore; inviata all’ascensore la persona. Al termine del ciclo, viene controllato che non sia scaduto il limite di tempo, nel caso il programma termina.

A questo punto, grazie ad “unlink” si elimina il socket, se esiste. Poi con ”bind” se ne crea uno nuovo e con “listen” si genera una coda di connessioni (massimo due).

*leggiArgomenti* al lancio del programma stabilisce la modalità con cui deve essere eseguito. Infine il “main” utilizza le system call “fork” che, a partire dal singolo processo “piani”, esegue due duplicazioni per creare i quattro piani (questi ultimi si duplicano in client e server). Infine li sincronizza, tramite il tempo di avvio, e fa in modo che attendano la terminazione del proprio figlio tramite l’istruzione “waitpid”.

```
1 #include <unistd.h> /* write, lseek, close, exit */
2 #include <sys/stat.h> /*open */
3 #include <fcntl.h> /*open*/
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <sys/socket.h>
7 #include <sys/un.h> /* Per socket AF_UNIX */
8 #include <time.h>
9 #include <string.h>
10 #include <errno.h>
11
12 #include "persona.h"
13 #include "linkedList.h"
14
15 #define DEFAULT_PROTOCOL 0
16 #define DURATA_MAX_MINUTI 5
17
18 static const int CONNESSIONE_PIANO_CLIENT = 1;
19 static const int CONNESSIONE_ASCENSORE = 0;
20
21
22 static const char* SOCKETS_PIANI[4] = { "piano0.sock", "piano1.sock",
23 "piano2.sock", "piano3.sock" };
24 static const char* PIANI_FILE_INPUT[4] =
25 { "piano0", "piano1", "piano2", "piano3" };
26 static const char* FILES_LOG[4] = { "piano0.log", "piano1.log",
27 "piano2.log", "piano3.log" };
28
29 time_t tempo_avvio;
30 time_t tempo_terminazione;
31
32 enum terminazione {
33     cinque_minuti, fine_servizio
34 } terminazione = cinque_minuti;
```

```

35 int numero_piano;
36
37
38 void scriviNelSocket(int SocketFd, const void* buffer, size_t size) {
39     int scritto = write(SocketFd, buffer, size);
40     if (scritto < size) {
41         char* msg;
42         asprintf(&msg,
43             "Errore invio messaggio socket \"%s\", terminazione al piano %i...\n",
44             SOCKETS_PIANI[numero_piano], numero_piano);
45         perror(msg);
46         exit(10);
47     }
48 }
49
50 void leggiDalSocket(int SocketFd, void* nuovo_arrivato, size_t size) {
51     int letto = read(SocketFd, nuovo_arrivato, size);
52     if (letto < 0) {
53         char* msg;
54         asprintf(&msg,
55             "Errore ricezione messaggio socket \"%s\", terminazione al piano %i...\n",
56             SOCKETS_PIANI[numero_piano], numero_piano);
57         perror(msg);
58         exit(10);
59     }
60 }
61
62 void client() {
63     char* categoria = NULL;
64
65     char* tempo_generazione = NULL;
66     int tempo_generazione_intero = 0;
67
68     char* destinazione = NULL;
69     int destinazione_intero = 0;
70
71     FILE * input_file_piani = NULL;
72
73     printf("Esecuzione del client, piano%i\n", numero_piano);
74
75     input_file_piani = fopen(PIANI_FILE_INPUT[numero_piano], "r");
76
77     if (input_file_piani == NULL) {
78         char* msg;
79         asprintf(&msg,
80             "Impossibile aprire file di input \"%s\", terminazione client, piano %i ",
81             PIANI_FILE_INPUT[numero_piano], numero_piano);
82         perror(msg);
83         exit(20);
84     }
85
86     while (1) {
87         char* riga_corrente = NULL;
88         int presente = 0;
89         int tempo;
90         size_t lunghezza = 0;
91
92         //legge una riga dal file di input e genera la persona
93         //se la riga e' vuota, termina
94
95         int bytes_letti=getline(&riga_corrente, &lunghezza, input_file_piani);
96         if(bytes_letti == -1){
97             char* msg;
98             asprintf(&msg,
99                 "Si e' verificato un errore di lettura \"%s\", terminazione client, piano %i\n",
100                PIANI_FILE_INPUT[numero_piano], numero_piano);
101             perror(msg);
102             exit(20);

```

```

103 }
104
105 if (strcmp(riga_corrente, "\n") == 0) {
106     printf(
107         "Ultima riga del file di input \"%s\", terminazione client, piano %i\n",
108         PIANI_FILE_INPUT[numero_piano], numero_piano);
109     break;
110 }
111
112 int SocketFd;
113 int SocketLenght;
114 struct sockaddr_un SocketAddress;
115 struct sockaddr* SocketAddrPtr;
116 SocketAddrPtr = (struct sockaddr*) &SocketAddress;
117 SocketLenght = sizeof(SocketAddress);
118
119 /* Create a UNIX socket, bidirectional, default protocol */
120 SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
121 if (SocketFd == -1) {
122     char* msg;
123     asprintf(&msg,
124         "Errore durante la creazione del socket \"%s\", terminazione client, piano %i ",
125         SOCKETS_PIANI[numero_piano], numero_piano);
126     perror(msg);
127     exit(10);
128 }
129
130 SocketAddress.sun_family = AF_UNIX; /* Set domain type */
131 strcpy(SocketAddress.sun_path, SOCKETS_PIANI[numero_piano]); /* Set name */
132
133 int connesso = connect(SocketFd, SocketAddrPtr, SocketLenght);
134 if (connesso == -1) {
135     char* msg;
136     asprintf(&msg, "Client al piano %i errore connessione", numero_piano);
137     perror(msg);
138     fclose(input_file_piani);
139     exit(12);
140 }
141 char* riga_temp = riga_corrente;
142
143 categoria = strsep(&riga_temp, " ");
144 tempo_generazione = strsep(&riga_temp, " ");
145 destinazione = strsep(&riga_temp, " ");
146
147 tempo_generazione_intero = atoi(tempo_generazione);
148 destinazione_intero = atoi(destinazione);
149
150 Persona persona = creaPersona(categoria[0], destinazione_intero);
151 free(riga_corrente);
152 tempo = time(NULL);
153 sleep(tempo_generazione_intero - (tempo - tempo_avvio));
154
155 scriviNelSocket(SocketFd, &CONNESSIONE_PIANO_CLIENT, sizeof(CONNESSIONE_PIANO_CLIENT));
156
157 long unsigned dimensione = sizeof(persona);
158 //invia la persona
159 scriviNelSocket(SocketFd, &persona, dimensione);
160
161 //invia la lunghezza della stringa
162 dimensione = strlen(persona.categoriaPersona) + 1;
163 scriviNelSocket(SocketFd, &dimensione, sizeof(dimensione));
164
165 //invia la stringa
166 scriviNelSocket(SocketFd, persona.categoriaPersona, dimensione);
167
168 close(SocketFd);
169 if (tempo_terminazione <= time(NULL)) {
170     printf("Durata programma raggiunta, terminazione client, piano %i\n",

```

```

171         numero_piano);
172         break;
173     }
174 }
175
176 fclose(input_file_piani);
177 }
178
179 void server() {
180     time_t ora;
181     lista_persone* coda = NULL;
182     nodo_lista_persone* testa = NULL;
183     FILE* log_file = NULL;
184     Persona* nuovo_arrivato = NULL;
185     int connessione = -1;
186
187     coda = creaListaPersone();
188     log_file = fopen(FILES_LOG[numero_piano], "w");
189
190     printf("Start server, piano%i\n", numero_piano);
191     if (log_file < 0) {
192         char* msg;
193         asprintf(&msg,
194             "Impossibile aprire file di log \"%s\", terminazione server piano %i...",
195             FILES_LOG[numero_piano], numero_piano);
196         perror(msg);
197         exit(20);
198     }
199
200     fprintf(log_file, "Avviato piano: %s (%i)\n",
201             ctime(&tempo_avvio),
202             (int) tempo_avvio);
203
204     unlink(SOCKETS_PIANI[numero_piano]);
205     int SocketFd;
206
207     socklen_t SocketLenght;
208     struct sockaddr_un SocketAddress;
209     struct sockaddr* SocketAddrPtr;
210
211     SocketAddrPtr = (struct sockaddr*) &SocketAddress;
212     SocketLenght = sizeof(SocketAddress);
213
214     /* Create a UNIX socket, bidirectional, default protocol */
215     SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
216     if (SocketFd == -1) {
217         printf(
218             "Errore durante la creazione del socket \"%s\", terminazione del server, piano %i...",
219             SOCKETS_PIANI[numero_piano], numero_piano);
220         exit(10);
221     }
222     SocketAddress.sun_family = AF_UNIX; /* Set domain type */
223     strcpy(SocketAddress.sun_path, SOCKETS_PIANI[numero_piano]); /* Set name */
224     int indirizzo = bind(SocketFd, SocketAddrPtr, SocketLenght);
225     if (indirizzo == -1) {
226         printf(
227             "Errore durante la bind del socket \"%s\", terminazione del server, piano %i...",
228             SOCKETS_PIANI[numero_piano], numero_piano);
229         perror("");
230         exit(10);
231     }
232     indirizzo = listen(SocketFd, 2);
233     if (indirizzo == -1) {
234         printf(
235             "Errore durante la listen del socket \"%s\", terminazione del server, piano %i...",
236             SOCKETS_PIANI[numero_piano], numero_piano);
237         perror("");
238         exit(10);
239     }

```

```

239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306

while (1) {
    int clientFd = accept(SocketFd, SocketAddrPtr, &SocketLength);
    if (clientFd == -1) {
        perror("Impossibile effettuare la connessione con i clients");
        printf("Terminazione del server, piano %i", numero_piano);
        exit(15);
    }
    //leggiDalSocket(clientFd, &connessione, sizeof(connessione));
    read(clientFd, &connessione, sizeof(connessione));

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

        nuovo_arrivato = (Persona*) malloc(sizeof(Persona));
        if (nuovo_arrivato == NULL) {
            printf(
                "Errore allocazione memoria, terminazione server piano %i...",
                numero_piano);
            exit(26);
        }

        leggiDalSocket(clientFd, nuovo_arrivato, sizeof(Persona));

        //legge lunghezza stringa categoriaPersona
        long unsigned dimensione = 0;
        leggiDalSocket(clientFd, &dimensione, sizeof(dimensione));

        //legge stringa categoriaPersona
        nuovo_arrivato->categoriaPersona = (char*) malloc(dimensione);
        leggiDalSocket(clientFd, nuovo_arrivato->categoriaPersona, dimensione);

        aggiungiPersonalista(coda, nuovo_arrivato);

        ora = time( NULL );
        printf(
            "[GENERATO] %s al piano %i, con destinazione piano %i, %s\n",
            nuovo_arrivato->categoriaPersona, numero_piano,
            nuovo_arrivato->destinazione, ctime(&ora));
        fprintf(log_file,
            "[GENERATO] %s, con destinazione piano %i, %s\n",
            nuovo_arrivato->categoriaPersona, nuovo_arrivato->destinazione,
            ctime(&ora));

    } else if (connessione == CONNESSIONE_ASCENSORE) { // si e' connesso l'ascensore
        printf("Connessione del server piano %i, connessione ascensore\n", numero_piano);
        int peso = 0;
        int presente = 0;
        //riceve il peso massimo caricabile dall'ascensore
        leggiDalSocket(clientFd, &peso, sizeof(int));
        while (1) {
            testa = getTestaLista(coda);
            if (testa == NULL) {
                presente = 0;
                scriviNelSocket(clientFd, &presente, sizeof(int));
                close(clientFd);
                break;
            }
            peso = peso - testa->persona->peso;
            if (peso < 0) {
                presente = 0;
                scriviNelSocket(clientFd, &presente, sizeof(int));
                close(clientFd);
                break;
            }
            //comunica all'ascensore che ci sono persone da inviare
            presente = 1;
            scriviNelSocket(clientFd, &presente, sizeof(int));
    }
}

```

```

307         //write(clientFd, &presente, sizeof(int));
308
309         //invia la persona
310         scriviNelSocket(clientFd, testa->persona, sizeof(Persona));
311         //write(clientFd, testa->persona, sizeof(Persona));
312
313         //invia la dimensione della stringa categoriaPersona
314         long unsigned dimensione = strlen(testa->persona->categoriaPersona) + 1;
315
316         scriviNelSocket(clientFd, &dimensione, sizeof(dimensione));
317
318         //invia la stringa categoriaPersona
319         scriviNelSocket(clientFd, testa->persona->categoriaPersona,
320                         dimensione);
321
322         eliminaPerTipo(coda, testa->persona->categoriaPersona);
323     }
324     close(clientFd);
325 } else {
326     printf("Errore durante la connessione");
327     continue;
328 }
329 if (tempo_terminazione <= time(NULL)) {
330     printf("Tempo limite raggiunto. ");
331     break;
332 }
333 close(SocketFd);
334 printf("Server terminato, piano%i\n", numero_piano);
335 ora = time(NULL);
336 fprintf(log_file, "Terminazione piano, %s (%i)\n", ctime(&ora), (int) ora);
337 fclose(log_file);
338 }
339
340
341 void leggiArgomenti(int argc, char* argv[]) {
342     if (argc == 1) {
343         printf("Esecuzione con tempo limite di 5 minuti\n");
344         return;
345     }
346     if (argc == 2 && strcmp(argv[1], "-termina") == 0) {
347         terminazione = fine_servizio;
348         printf(
349             "Esecuzione senza tempo limite, fino alla fine del servizio\n");
350     } else {
351         printf(
352             "Uso:\n%s: Il programma termina dopo 5 minuti dall'avvio.\n"
353             "%s -termina: Il programma prosegue fino a quando non ha finito di servire i
354             passeggeri.\n",
355             argv[0], argv[0]);
356         exit(1);
357     }
358 }
359
360 int main(int argc, char * argv[]) {
361     int status = 0;
362     numero_piano = 0;
363     int pidserver1 = 0;
364     int pidserver2 = 0;
365     int pidserver3 = 0;
366
367     leggiArgomenti(argc, argv);
368
369     tempo_avvio = time(NULL) + 3;
370     int prima_fork = 0;
371     int pid = fork();
372     if (pid) {
373         prima_fork++;
374     }

```

```

374     pidserver2 = pid;
375
376     pid = fork();
377     //assegna i numeri dei piani differenziandoli in base ai risultati delle fork
378     //e salva anche i pid dei figli (che saranno i piani server) per la terminazione
379     if (pid) {
380         if (prima_fork) {
381             numero_piano = 2;
382             pidserver3 = pid;
383         } else {
384             numero_piano = 0;
385             pidserver1 = pid;
386         }
387     } else {
388         if (prima_fork) {
389             numero_piano = 3;
390         } else {
391             numero_piano = 1;
392         }
393     }
394
395     pid = fork();
396     time_t now = time(NULL);
397     tempo_terminazione = now + (DURATA_MAX_MINUTI * 60);
398     // se il pid == 0, quindi è un nuovo server allora esso sarà un client, altrimenti
399     // server
400     if (!pid) {
401         sleep(tempo_avvio - now + 2);
402         tempo_avvio = time(NULL);
403         client();
404     } else {
405         sleep(tempo_avvio - now);
406         tempo_avvio = time(NULL) + 2;
407         server();
408
409         //aspetta terminazione del piano client corrispondente
410         waitpid(pid, &status, 0);
411     }
412     //per server piano 2 risulta pidserver3 != 0 e aspetta che server piano 3 termini
413     if (pidserver3)
414         waitpid(pidserver3, &status, 0);
415     //per server piano 0 risulta pidserver1 != 0 e aspetta che server piano 1 termini
416     if (pidserver1)
417         waitpid(pidserver1, &status, 0);
418     //per server piano 0 risulta pidserver2 != 0 e aspetta che server piano 2 termini
419     if (pidserver2)
420         waitpid(pidserver2, &status, 0);
421     return status;
}

```

---

“ascensore.c” include “persona.h” e “linkedlist.h”, definisce i valori: “DEFAULT\_PROTOCOL”; “TEMPO\_SOSTA”; “TEMPO\_SPOSTAMENTO”; “PESO\_MASSIMO”. Quindi vari enumeratori (per direzioni, piani e terminazioni) e la costante relativa alla connessione dell’ascensore seguita dall’array di stringhe gli identificatori dei sockets dei vari piani. Infine tre variabili (“contatore\_bambini”, “contatore\_adulti” e “contatore\_addetti\_consegne”) utilizzate nel resoconto dell’attività giornaliera.

*spostamentoAscensore* si occupa di muovere l’ascensore rispettando le specifiche.

*scriviNelSocket* e *leggiDalSocket*, analoghe alle omonime in “piani.c”.

*salitaPersone* tramite il descrittore del socket e il puntatore al file di Log (utilizzato per annotare l’arrivo delle persone), restituisce il numero di persone caricate sull’ascensore. A tal proposito controlla la presenza di almeno una persona e in caso affermativo, la aggiunge alla lista delle persone sull’ascensore. Quindi stampa tutte le informazioni relative a quest’ultima, salvandole nel file di Log, e aggiorna il peso e il numero di persone a bordo.

*discesaPersone* utilizza il file di Log ed implementa la discesa di una persona (ne libera la memoria). Tiene traccia delle persone servite aggiornando gli opportuni contatori.

*main* si occupa di creare la lista delle persone presenti in ascensore, quindi apre il file di Log, e attenderà (comunicando tramite socket con il piano server 0) ciclicamente la partenza del suddetto. Se riesce a connettersi, comunica al server il codice dell’ascensore, e dopo aver calcolato il la capienza massima dell’ascensore la comunica al piano server. A questo punto, tramite “*salitaPersone*”, considera il numero di persone entrate nell’ascensore e attende 10 ms dopo aver chiuso il socket.

Nel caso in cui almeno una persona sia salita, esce dal ciclo ed entra in quello principale, dove:

1. Viene richiamata “*spostamentoAscensore*”, ed aspetta 6 secondi (tempo di spostamento + tempo di sosta);
2.
  - a) Tramite *discesaPersone*, si istanzia la socket per tentare di comunicare con il piano server (in caso di fallimento agisce sull’array dei piani al fine di notificare la terminazione del piano corrispondente). Infine, controlla che il valore di “piani\_non\_terminati” sia uguale a 0, se così è, termina l’ascensore;
  - b) Se riesce invece a connettersi: scrive sul socket il codice dell’ascensore; calcola e invia il peso rimanente e richiama *salitaPersone*. Infine chiude la socket e, fuori dal ciclo, il file di Log.

---

```

1 #include <sys/socket.h>
2 #include <sys/un.h> /* Per socket AF_UNIX */
3 #include <time.h>
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <string.h>
7 #include <errno.h>
8 #include <unistd.h>
9
10 #include "persona.h"
11 #include "linkedList.h"
12
13 #define PESO_MASSIMO 300
14 #define DEFAULT_PROTOCOL 0
15 #define TEMPO_SOSTA 3
16 #define TEMPO_SPOSTAMENTO 3
17
18 static const int CONNESSIONE_ASCENSORE = 0;
19
20 enum terminazione {
21     cinque_minuti, fine_file
22 } terminazione;
23
24 enum direzione {
25     ALTO, BASSO
26 } direzione;
27
28 enum piano {
29     piano0, piano1, piano2, piano3
30 } piano;
31
32
33 static const char* SOCKETS_PIANI[4] = { "piano0.sock", "piano1.sock",
34 "piano2.sock", "piano3.sock" };
35
36 lista_persone* lista = NULL;
37 int carico = 0;
38 time_t tempo_avvio;
39 int contatore_bambini = 0;
40 int contatore_adulti = 0;
41 int contatore_addetti_consegne = 0;
42
43 void spostamentoAscensore() { // L'ascensore si sposta di un piano in base alla direzione corrente
44 if (direzione == ALTO) {
45     if (piano == piano3) {
46         direzione = BASSO;
47         piano--;
48     } else {
49         piano++;
50     }
51 } else {
52     if (piano == piano0) {
53         direzione = ALTO;
54         piano++;
55     } else {
56         piano--;
57     }
58 }
59 }
60
61 void scriviNelSocket(int SocketFd, const void* buffer, size_t size) { // Come in piani.c
62     int scritto = write(SocketFd, buffer, size);
63     if (scritto < size) {
64         char* msg;
65         asprintf(&msg, "Errore durante l'invio, piano %i, terminazione ascensore ",
66         piano);
67         perror(msg);
68         exit(10);

```

---

```

69     }
70 }
71
72 void leggiDalSocket(int SocketFd, void* nuovo_arrivato, size_t size) { // Come in piano.c
73     int letto = read(SocketFd, nuovo_arrivato, size);
74     if (letto < 0) {
75         char* msg;
76         asprintf(&msg, "Errore durante la ricezione, piano %i, terminazione ascensore ",
77                  piano);
78         perror(msg);
79         exit(10);
80     }
81 }
82
83 int salitaPersone(int SocketFd, FILE* logFp) { // Si occupa di aspettare le persone e registrarne i dati nel file di
84     * LOG, aggiorna l'ascensore
85     int persona_presente = 0;
86
87     leggiDalSocket(SocketFd, &persona_presente, sizeof(int));
88     if (!persona_presente) {
89         return 0;
90     }
91
92     Persona* nuovo_arrivato = (Persona*) malloc(sizeof(Persona));
93     int contatore_caricate = 0;
94     while (persona_presente) {
95         nuovo_arrivato = (Persona*) malloc(sizeof(Persona));
96         //printf("ricezione persona");
97         leggiDalSocket(SocketFd, nuovo_arrivato, sizeof(Persona));
98         aggiungiPersonaLista(lista, nuovo_arrivato);
99
100        //legge dimensione stringa categoriaPersona
101        long unsigned dimensione = 0;
102        //printf("ricezione dimensione");
103        leggiDalSocket(SocketFd, &dimensione, sizeof(dimensione));
104        nuovo_arrivato->categoriaPersona = (char*) malloc(dimensione);
105
106        //legge la stringa nome tipo
107        //printf("ricezione tipo");
108        leggiDalSocket(SocketFd, nuovo_arrivato->categoriaPersona, dimensione);
109
110        time_t ora = time( NULL );
111
112        printf(
113            "[SALITA] %s al piano %i, con destinazione %i, %s\n",
114            nuovo_arrivato->categoriaPersona, piano, nuovo_arrivato->destinazione,
115            ctime(&ora));
116        fprintf(logFp,
117            "[SALITA] %s al piano %i, con destinazione %i, %s\n",
118            nuovo_arrivato->categoriaPersona, piano, nuovo_arrivato->destinazione,
119            ctime(&ora));
120
121        carico = carico + nuovo_arrivato->peso;
122        contatore_caricate++;
123
124        leggiDalSocket(SocketFd, &persona_presente, sizeof(int));
125    }
126    return contatore_caricate;
127 }
128
129 void discesaPersone(FILE* logFp) { // Come carica persone, ma ne cancella i dati per simulare l'arrivo a
130     * destinazione
131     Persona* scesa = NULL;
132     scesa = eliminaDiscesa(lista, piano);
133     while (scesa != NULL) {
134         time_t ora = time( NULL );
135         printf("[DISCESA] %s al piano %i, %s\n",
136             scesa->categoriaPersona, piano, ctime(&ora));

```

```

135     fprintf(logFp, "[DISCESA] %s al piano %i, %s\n",
136     scesa->categoriaPersona, piano, ctime(&ora));
137
138     switch(scesa->peso){
139         case 80:
140             contatore_adulti++;
141             break;
142         case 40:
143             contatore_bambini++;
144             break;
145         case 90:
146             contatore_adetti_consegne++;
147             break;
148     }
149     carico = carico - scesa->peso;
150     free(scesa);
151     scesa = eliminaDiscesa(lista, piano);
152 }
153 }
154
155 int main(int argc, char *argv[]) { // Comunica con il Socket- si autentica con il Server - Carica e Scarica persone
156     short piani_terminati[4] = { 0, 0, 0, 0 };
157     int piani_non_terminati = 4;
158     int peso_massimo_imbarcabile = PESO_MASSIMO;
159     piano = 0;
160
161     lista = creaListaPersone();
162
163     sleep(4);
164     tempo_avvio = time(NULL);
165
166     FILE* logFp = NULL;
167     logFp = fopen("ascensore.log", "w");
168     fprintf(logFp, "Avvio ascensore: %s\n", ctime(&tempo_avvio));
169
170     int persone_caricate = 0;
171     do {
172         int SocketFd, SocketLenght, tempo;
173         struct sockaddr_un SocketAddress;
174         struct sockaddr* SocketAddrPtr;
175
176         SocketAddrPtr = (struct sockaddr*) &SocketAddress;
177         SocketLenght = sizeof(SocketAddress);
178
179         /* Create a UNIX socket, bidirectional, default protocol */
180         SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
181         SocketAddress.sun_family = AF_UNIX; /* Set domain type */
182         strcpy(SocketAddress.sun_path, SOCKETS_PIANI[piano]); /* Set name */
183         int connesso = connect(SocketFd, SocketAddrPtr, SocketLenght);
184
185         if (connesso == -1) {
186             char* msg;
187             asprintf(&msg, "Ascensore NON CONNESSO tramite socket \"%s\"\n",
188             SOCKETS_PIANI[piano]);
189             perror(msg);
190             close(SocketFd);
191             exit(21);
192         }
193
194         scriviNelSocket(SocketFd, &CONNESSIONE_ASCENSORE,
195             sizeof(CONNESSIONE_ASCENSORE));
196
197         peso_massimo_imbarcabile = PESO_MASSIMO - carico;
198         scriviNelSocket(SocketFd, &peso_massimo_imbarcabile, sizeof(int));
199
200         persone_caricate = salitaPersone(SocketFd, logFp);
201         close(SocketFd);
202         usleep(10000);
203     } while (persone_caricate == 0);

```

```

203     printf("Carico dell'ascensore = %i\n", carico);
204
205     while (1) {
206         spostamentoAscensore();
207         sleep(TEMPO_SPOSTAMENTO);
208         time_t ora = time(NULL);
209         printf("[FERMATA] Piano %i, %s\n", piano, ctime(&ora));
210         sleep(TEMPO_SOSTA);
211         discesaPersone(logFp);
212         int SocketFd, SocketLenght, tempo;
213         struct sockaddr_un SocketAddress;
214         struct sockaddr* SocketAddrPtr;
215
216         SocketAddrPtr = (struct sockaddr*) &SocketAddress;
217         SocketLenght = sizeof(SocketAddress);
218
219         /* Create a UNIX socket, bidirectional, default protocol */
220         SocketFd = socket(AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
221         SocketAddress.sun_family = AF_UNIX; /* Set domain type */
222         strcpy(SocketAddress.sun_path, SOCKETS_PIANI[piano]); /* Set name */
223         int connesso = connect(SocketFd, SocketAddrPtr, SocketLenght);
224
225         if (connesso != 0) {
226             if (piani_terminati[piano]) {
227                 if (piani_non_terminati == 0) {
228                     close(SocketFd);
229                     break;
230                 }
231                 continue;
232             } else {
233                 piani_non_terminati--;
234                 piani_terminati[piano] = 1;
235                 continue;
236             }
237         } else {
238             printf("Connessione tramite \"%s\"\n", SOCKETS_PIANI[piano]);
239         }
240
241         scriviNelSocket(SocketFd, &CONNESSIONE_ASCENSORE,
242             sizeof(CONNESSIONE_ASCENSORE));
243
244         peso_massimo_imbarcabile = PESO_MASSIMO - carico;
245         scriviNelSocket(SocketFd, &peso_massimo_imbarcabile, sizeof(int));
246
247         persone_caricate = salitaPersone(SocketFd, logFp);
248         printf("Carico dell'ascensore = %i\n", carico);
249         close(SocketFd);
250     }
251
252     printf("Terminazione ascensore...\n");
253     time_t tempo_terminazione = time(NULL);
254     fprintf(logFp, "Terminazione ascensore %s (%i)\n",
255             ctime(&tempo_terminazione), (int) tempo_terminazione);
256     char* msg;
257     asprintf(&msg, "Resoconto attivita giornaliera:\nPersonne servite %i\nBambini %i\nAdulti %i\nAddetti alle
258     consegne %i\n",
259     contatore_bambini + contatore_adulti + contatore_addetti_consegne, contatore_bambini, contatore_adulti,
260     contatore_addetti_consegne);
261     printf("%s", msg);
262     fprintf(logFp, "%s", msg);
263     fclose(logFp);
264     return 0;
265 }
```

---

“**makefile**” include, oltre ai target per la compilazione, i seguenti:

1. “**cleansocket**” cancella i file “.sock”;
2. “**cleanlog**” elimina i file “.log”;
3. “**clean**” cancella gli eseguibili e i codici oggetto generati in fase di compilazione;
4. “**cleanall**” utilizzato per eseguire tutte le operazioni di cancellazione in una sola volta.

```
1 all: ascensore piani
2
3 piani: piani.c linkedList.o persona.o linkedList.h
4     cc piani.c linkedList.o persona.o -o piani
5
6 ascensore: ascensore.c persona.o linkedList.o
7     cc ascensore.c linkedList.o persona.o -o ascensore
8
9 linkedList.o: linkedList.c linkedList.h
10    cc -c linkedList.c -o linkedList.o
11
12 persona.o: persona.c persona.h
13     cc -c persona.c -o persona.o
14 cleanall: clean cleansocket cleanlog
15
16 clean:
17     rm -f ascensore piani
18     rm -f linkedList.o persona.o
19 cleansocket:
20     rm -f piano0.sock piano1.sock piano2.sock piano3.sock
21 cleanlog:
22     rm -f piano0.log piano1.log piano2.log piano3.log ascensore.log
23
```

---

## ESECUZIONE TIPO

```
[Daniels-MacBook-Pro:sources Danny$ make all
cc -c linkedList.c -o linkedList.o
cc ascensore.c linkedList.o persona.o -o ascensore
cc piani.c linkedList.o persona.o -o piani
```

Iniziamo con il comando “make all” che tramite il *makefile* genera i file eseguibili necessari.

N.B.: La shell con sfondo nero si riferisce all’eseguibile *piani*. Mentre la shell con sfondo blu all’eseguibile *ascensore*.

```
[Daniels-MacBook-Pro:sources Danny$ ./piani
Esecuzione con tempo limite di 5 minuti
Start server, piano3
Start server, piano0
Start server, piano1
Start server, piano2
Esecuzione del client, piano2
Esecuzione del client, piano3
Esecuzione del client, piano1
Esecuzione del client, piano0
```

Tramite il comando “./piani” vengono eseguiti i vari server e client necessari per il funzionamento del progetto.

```
Connessione server piano 0, con piano client
[GENERATO] Bambino al piano 0,con destinazione piano 3, Mon Feb 13 18:54:58 2017

Connessione del server piano 0, connessione ascensore
```

Il server del piano 0 si connette al rispettivo client, quest’ultimo genera un bambino (i cui dettagli sono visibili nella seconda riga dello screen). Quindi l’ascensore si connette al server.

```
Daniels-MacBook-Pro:sources Danny$ ./ascensore
[SALITA] Bambino al piano 0, con destinazione 3, Mon Feb 13 19:09:23 2017
Carico dell'ascensore = 40
[FERMATA] Piano 1, Mon Feb 13 19:09:26 2017
Connessione tramite "piano1.sock"
```

L'ascensore si connette al server del piano 0, il bambino sopra generato “sale” sull'ascensore e conseguentemente il carico di quest'ultimo si incrementa di 40 (peso del bambino). Quindi l'ascensore si ferma al piano successivo e si connette alla relativa socket.

```
[FERMATA] Piano 3, Mon Feb 13 19:14:32 2017
[DISCESA] Bambino al piano 3, Mon Feb 13 19:14:35 2017
```

Appena l'ascensore raggiunge il piano 3 il bambino effettua la discesa.

Questi passi saranno ripetuti fino al raggiungimento del tempo limite, qualora si sia scelta questa modalità.

```
Terminazione ascensore...
Resoconto attività giornaliera:
Persone servite 72
Bambini 30
Adulti 26
Addetti alle consegne 16
```

Una volta terminata la sua corsa l'ascensore termina la sua esecuzione dopo aver stampato il resoconto dell'attività giornaliera.

```
Tempo limite raggiunto. Server terminato, piano1
Connessione del server piano 2, connessione ascensore
Tempo limite raggiunto. Server terminato, piano2
Connessione del server piano 3, connessione ascensore
Tempo limite raggiunto. Server terminato, piano3
Connessione del server piano 0, connessione ascensore
Tempo limite raggiunto. Server terminato, piano0
```

Come l'ascensore, anche i piani server vengono terminati uno ad uno.

## Di seguito proponiamo alcuni situazioni atipiche:

- Se eseguito soltanto il file *ascensore* verrà visualizzato il seguente errore:

```
Daniels-MacBook-Pro:sources Danny$ ./ascensore
Ascensore NON CONNESSO tramite socket "piano0.sock"
: Connection refused
```

- Se durante l'esecuzione l'ascensore non è più in grado di comunicare con i vari server esso porterà a termine la sua corsa. Una volta che ogni passeggero presente al suo interno ha raggiunto la propria destinazione, termina.

```
Daniels-MacBook-Pro:sources Danny$ ./ascensore
[SALITA] Bambino al piano 0, con destinazione 3, Mon Feb 13 19:44:13 2017
Carico dell'ascensore = 40
[FERMATA] Piano 1, Mon Feb 13 19:44:16 2017

Connessione tramite "piano1.sock"
[SALITA] Adatto al piano 1, con destinazione 0, Mon Feb 13 19:44:22 2017
[SALITA] Bambino al piano 1, con destinazione 0, Mon Feb 13 19:44:22 2017
[SALITA] Adatto al piano 1, con destinazione 2, Mon Feb 13 19:44:22 2017
Carico dell'ascensore = 240
[FERMATA] Piano 2, Mon Feb 13 19:44:25 2017
Disconnessione Server
[DISCESA] Adatto al piano 2, Mon Feb 13 19:44:28 2017
[FERMATA] Piano 3, Mon Feb 13 19:44:31 2017
[DISCESA] Bambino al piano 3, Mon Feb 13 19:44:34 2017
[FERMATA] Piano 2, Mon Feb 13 19:44:37 2017
[FERMATA] Piano 1, Mon Feb 13 19:44:43 2017
[FERMATA] Piano 0, Mon Feb 13 19:44:49 2017
[DISCESA] Adatto al piano 0, Mon Feb 13 19:44:52 2017
[DISCESA] Bambino al piano 0, Mon Feb 13 19:44:52 2017
[FERMATA] Piano 1, Mon Feb 13 19:44:55 2017
Terminazione ascensore...
Resoconto attività giornaliera:
Persone servite 4
Bambini 2
Adulti 2
```