RELAZIONE SISTEMI OPERATIVI 2023/2024

Docente: Prof. Francesco Quaglia

Macroarea Ingegneria

a cura di Andrea Tosi, 0327864

Sommario

Specifiche del progetto

Servizio di messaggistica

Realizzazione di un servizio di scambio messaggi supportato tramite un server sequenziale o concorrente (a scelta). Il servizio deve accettare messaggi provenienti da client (ospitati in generale su macchine distinte da quella dove riese il server) ed archiviarli.

L'applicazione client deve fornire ad un utente le seguenti funzioni:
1. Lettura tutti i messaggi spediti all'utente.
2. Spedizione di un nuovo messaggio a uno qualunque degli utenti del sistema.
3. Cancellare dei messaggi ricevuti dall'utente.

Un messaggio deve contenere almeno i campi Destinatario, Oggetto e Testo.

Si precisa che la specifica prevede la realizzazione sia dell'applicazione client che di quella server. Inoltre, il servizio potra' essere utilizzato solo da utenti autorizzati (deve essere quindi previsto un meccanismo di autenticazione).

Descrizione generale

Lista

Per conservare i messaggi inviati, per ogni utente esiste una lista collegata singolarmente con head e tail che, nonostante la semplicità della sua implementazione, consente di avere tempi di inserimento alla fine della lista dell'ordine di O(1).

Mutex

Siccome il server è multithread, per garantire l'accesso esclusivo a risorse condivise tra più thread sono stati utilizzati due mutex: uno per evitare di corrompere il file pointer di "passwd" e uno per la corretta gestione delle liste dei messaggi.

Segnali

Nel client, tramite signal() sono ignorati SIGINT, SIGQUIT e SIGTERM in modo da interrompere l'esecuzione dell'applicazione solo tramite input intero (la chiusura della connessione con il server deve superare la fase di autenticazione). Invece, il segnale SIGPIPE, catturato quando si rompe la connessione tra client e server inaspettatamente, viene gestito chiudendo il socket del client e terminando il processo.

Il server può essere chiuso tramite i segnali SIGINT, SIGTERM, SIGQUIT e SIGHUP che attivano un handler che elimina dalla memoria le liste di messaggi di tutti i client che erano connessi. Invece, nel caso di SIGPIPE, viene semplicemente terminato il thread relativo al client che ha provocato la rottura della pipe.

File “passwd”

Ogni riga del file è del tipo USERNAME:ENCRYPTED_PASSWORD\n (perciò lo username non può contenere il carattere ‘:’). La funzione crittografica di hash adottata è SHA-256.

Socket

Per la comunicazione sono stati utilizzati socket AF_INET via IPV4 e TCP tra il server e i client.

Libreria termios.h

Questa libreria serve a modificare la configurazione del terminale in uso. È utilizzata per nascondere l’input (disabilitando l’echo dei caratteri immessi da tastiera) ogni volta che il client chiede di scrivere la password.

Server

(N.B. ho scelto di sviluppare un server concorrente che fa uso di un thread per ogni client connesso perché più efficiente)

Al suo lancio, il server apre un file “passwd” dove verranno conservati username e password criptata relativa e crea un socket che viene messo in stato listening per accogliere eventuali connessioni dai client che desiderano usufruire del servizio. Ogni volta che un client si collega, viene creato un thread che preleva dal client username e password dell’utente e, dopo aver criptato quest’ultima tramite la funzione crittografica di hash SHA-256, li registra all’interno del file “passwd”.

Al temine della registrazione dell’utente, il server riceve dal client un numero intero da 1 a 5, che corrisponde a ciò che l’utente vuole fare. In ogni caso, prima di eseguire ciò che l’utente desidera, occorre che quest’ultimo venga autenticato. La funzione autenticazione() riceve la password acquisita dal client, la cripta e la confronta con quella salvata in “passwd”.

Per scrivere un messaggio, il server riceve il nome dell’utente che vuole scrivere il messaggio, poi viene chiesto all’utente di immettere in input il destinatario da contattare; perciò, vengono prima stampati gli utenti disponibili, che sono quelli connessi al server. Successivamente, il server riceve dal client il destinatario e verifica se esso si trova in “passwd”. In caso negativo viene rieseguita la stessa parte di codice finché il server non ottiene un destinatario accettabile. Dopodiché vengono ricevuti oggetto e testo del messaggio da inviare, che viene inserito alla fine della lista relativa al destinatario.

Per leggere tutti i messaggi inviati all’utente, basta scorrere tutta la lista stampando il contenuto di tutti i nodi, mentre per cancellarli basta eseguirne la free().

Infine, per terminare la connessione con un determinato client, occorre cancellare da “passwd” la riga relativa a quel client ed eseguire la free() dei messaggi inviati a tale client.

Per chiudere il server, basta mandare un segnale come SIGINT, SIGTERM o SIGQUIT.

N.B.: nel caso in cui il server non riceva un input dall’utente entro 30 minuti, il client verrà terminato.

Client

Al suo lancio, il client crea un socket per connettersi al server ed esegue la registrazione dell'utente, ossia prende in input lo username (deve essere diverso dagli altri username contenuti in "passwd") e la password, che poi invia al server. Successivamente, parte un loop: il client stampa le opzioni che il servizio offre e prende in input un numero intero da 1 a 5 corrispondente all'opzione scelta e lo invia al server. Tuttavia, prima di esaudire la richiesta il client autentica l'utente chiedendogli la password. Solo in caso affermativo, verrà eseguito il resto del codice, altrimenti si riparte con il loop.

Limiti utenti connessi al server: 150. Nel caso in cui un ulteriore client volesse connettersi, rimarrà in attesa che si liberi un posto (ossia che un client già connesso al server si disconnetta).

Modalità d'uso

Manuale di compilazione:

È stato preparato un Makefile per la compilazione, che permette i seguenti comandi:

- make all genera gli eseguibili di server e client
- make server genera l'eseguibile del server
- make client genrea l'eseguibile del client
- make clean rimuove gli eseguibili di server e client

Manuale del server:

Il server, una volta avviato non ha bisogno di input, a parte quello dei segnali per interromperlo (^C, ^\, ecc.).

Manuale del client:

Il client, una volta avviato, chiederà username e password per registrarsi, dopodiché chiederà cosa vuole fare l'utente, che deve inserire un numero da 1 a 5:

- 1 per leggere i messaggi a lui inviati;
- 2 per spedire un messaggio a un utente connesso al server (anche a sé stesso);
- 3 per cancellare tutti i messaggi a lui inviati;
- 4 per stampare gli username di tutti gli utenti connessi al server;
- 5 per disconnettersi.

Viene richiesta la password per identificarsi ogni volta che si vuole usufruire di uno dei suddetti servizi.

Codice sorgente

lib.h

```
#include <stdlib.h>
#include <stdio.h>
#include <termios.h> //novità
#include <crypt.h> //novità
```

```
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <time.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <pthread.h>
#include <signal.h>
#include <errno.h>
#include <string.h>
#include <sys/sem.h>

#define PORT 5001
#define MAX_CLIENT 150
#define PENDING 100
#define TIMEOUT 1800//secondi (30 minuti)

void TIMEOUT_OCCURRED(int i);

#define CORSIVO "\033[3m"
#define RESET "\033[0m"

#define CLEAR_INPUT_BUFFER while(getchar() != '\n') //da usare solo dopo scanf("%ms",...); o altre funzioni di input che ignorano '\n'


struct messaggio{
    char mittente[129];
    char destinatario[129];
    char *oggetto;
    char *testo;
    struct messaggio *next;
};
//archiviazione messaggi tramite array di puntatori (ogni entry dell'array individua una lista di messaggi relativi al client corrispondente)


void no_echo_input(struct termios *original){
        struct termios term_conf;

    //cambio impostazioni terminale in modo da non eseguire la echo dei caratteri trasmessi su stdin
    tcgetattr(STDIN_FILENO, original);
    term_conf = *original;
    term_conf.c_lflag &= ~(ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &term_conf);
}


void reset_echo_input(struct termios *original){
```

```
    //reset impostazioni terminale in modo che venga eseguita la echo dei caratteri trasmessi su
stdin
    tcsetattr(STDIN_FILENO, TCSANOW, original);
}
```

server.c

```
#include "lib.h"

FILE *passwd;
int max_clients = MAX_CLIENT;
char username[MAX_CLIENT][129] = {""};
int sock_des[MAX_CLIENT]; //array di descrittori di socket server connessi a client creati da
connect() (valgono -1 nel caso in cui non ci sia un client nell'entry considerato)
pthread_t tid[MAX_CLIENT] = {0}; //array di tid dei thread creati nel main, ognuno relativo ad un
client connesso
struct list{
        struct messaggio *head;
        struct messaggio *tail;
};
pthread_mutex_t mutex_file = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_mess = PTHREAD_MUTEX_INITIALIZER;
struct list head_list[MAX_CLIENT];



void termina_connessione(int i);


//ritorna -1 se non la trova, altrimenti ritorna la posizione del file pointer che indica l'inizio della
parola (usato per cercare usernames in passwd)
int cerca_username_in_file(FILE *file, char *word){
    int file_pointer;
    char parola[129];

    pthread_mutex_lock(&mutex_file);

    fseek(file, 0, SEEK_SET);
    while(fscanf(file, "%[^:]s", parola) != EOF){
        if(strcmp(parola, word) == 0){
            file_pointer = ftell(file) - strlen(word);
            fseek(file, 0, SEEK_END);
            pthread_mutex_unlock(&mutex_file);

            return file_pointer;
        } else {
            fseek(file, 65, SEEK_CUR);
            continue;
        }
    }
```

```
    pthread_mutex_unlock(&mutex_file);
    return -1;
}



void print_users(int i){
    char user[129], *line = NULL;
    int check, size;
    size_t len;

    pthread_mutex_lock(&mutex_file);

    fseek(passwd, 0, SEEK_SET);

    while((check = getline(&line, &len, passwd)) != -1){
            sscanf(line, "%[^:]", user);
            size = strlen(user) + 1;
            while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
                    while(send(sock_des[i], user, size, 0) == -1   &&   errno == EINTR);
    }
            free(line);
            size = -1;
            while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);

    pthread_mutex_unlock(&mutex_file);
}



int autenticazione(int i){
    char *buffer, encrypted_password[64];
    int size, file_pointer, res;

            //acquisizione password da confrontare con quella contenuta in passwd
    while(recv(sock_des[i], &size, sizeof(int), 0) == -1){
            if(errno == EAGAIN || errno == EWOULDBLOCK){
                    TIMEOUT_OCCURRED(i);
                    }else if(errno == EINTR){
                            continue;
                    }else{
                            printf("recv failed, errno: %s\n", strerror(errno));
                            exit(EXIT_FAILURE);
                    }
            }
    if((buffer = malloc(size)) == NULL){
       printf("malloc failed, errno: %s\n", strerror(errno));
       exit(EXIT_FAILURE);
    }
    while(recv(sock_des[i], buffer, size, 0) == -1){
            if(errno == EAGAIN || errno == EWOULDBLOCK){
```

```
                TIMEOUT_OCCURRED(i);
                }else if(errno == EINTR){
                        continue;
                }else{
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                }
        }

        //acquisizione password contenuta in passwd
    file_pointer = cerca_username_in_file(passwd, username[i]);

    pthread_mutex_lock(&mutex_file);

    fseek(passwd, file_pointer + strlen(username[i]) + 1, SEEK_SET);
    fgets(encrypted_password, 64, passwd);
    fseek(passwd, 0, SEEK_END);

    pthread_mutex_unlock(&mutex_file);

        //confronto password
    if(strcmp(crypt(buffer, "$5$hakunamatataraga"), encrypted_password) == 0){
        res = 1;
        while(send(sock_des[i], &res, sizeof(int), 0) == -1   &&   errno == EINTR);
                free(buffer);
        return 1;
    }else{
        res = 0;
        while(send(sock_des[i], &res, sizeof(int), 0) == -1   &&   errno == EINTR);
                free(buffer);
        return 0;
    }
}



void inserisci_mess_in_lista(struct messaggio *msg, int i){
    struct messaggio *curr;

    if(head_list[i].head == NULL){
        head_list[i].head = msg;
        head_list[i].tail = msg;
    }else{
            head_list[i].tail -> next = msg;
        head_list[i].tail = msg;
    }
}



void spedisci_mess(int i){
```

```
   struct messaggio *mess_ptr;
   int index = 0, size, found_dest = 0;

   if((mess_ptr = malloc(sizeof(struct messaggio))) == NULL){
      printf("malloc failed, errno: %s\n", strerror(errno));
      exit(EXIT_FAILURE);
   }

         //acquisizione mittente
   while(recv(sock_des[i], mess_ptr -> mittente, 129, 0) == -1){
                  if(errno == EAGAIN || errno == EWOULDBLOCK){
                        free(mess_ptr);
                  TIMEOUT_OCCURRED(i);
                  }else if(errno == EINTR){
                        continue;
                  }else{
                        free(mess_ptr);
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                  }
         }

         //acquisizione destinatario
         do{
                  print_users(i);
         while(recv(sock_des[i], mess_ptr -> destinatario, 129, 0) == -1){
                        if(errno == EAGAIN || errno == EWOULDBLOCK){
                              free(mess_ptr);
                        TIMEOUT_OCCURRED(i);
                        }else if(errno == EINTR){
                              continue;
                        }else{
                              free(mess_ptr);
                              printf("recv failed, errno: %s\n", strerror(errno));
                              exit(EXIT_FAILURE);
                        }
         }
                  if(cerca_username_in_file(passwd, mess_ptr -> destinatario) == -1){
                        size = strlen("username non trovato") + 1;
                  while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
                  while(send(sock_des[i], "username non trovato", strlen("username non trovato") + 1,
0) == -1   &&   errno == EINTR);
                  }else{
                        found_dest = 1;
                        size = strlen("username trovato") + 1;
                        while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
                        while(send(sock_des[i], "username trovato", strlen("username trovato") + 1,
0) == -1   &&   errno == EINTR);
                  }
         }while(!found_dest);
   while(strcmp(username[index], mess_ptr -> destinatario) != 0){
```

```
    index++;
}

        //acquisizione oggetto del messaggio
while(recv(sock_des[i], &size, sizeof(int), 0) == -1){
        if(errno == EAGAIN || errno == EWOULDBLOCK){
                free(mess_ptr);
                TIMEOUT_OCCURRED(i);
                }else if(errno == EINTR){
                        continue;
                }else{
                        free(mess_ptr);
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                }
        }
if((mess_ptr -> oggetto = malloc(size)) == NULL){
                printf("malloc failed, errno: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}
while(recv(sock_des[i], mess_ptr -> oggetto, size, 0) == -1){
                if(errno == EAGAIN || errno == EWOULDBLOCK){
                        free(mess_ptr -> oggetto);
                        free(mess_ptr);
                TIMEOUT_OCCURRED(i);
                }else if(errno == EINTR){
                        continue;
                }else{
                        free(mess_ptr -> oggetto);
                        free(mess_ptr);
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                }
        }

        //acquisizione testo del messaggio
while(recv(sock_des[i], &size, sizeof(int), 0) == -1){
        if(errno == EAGAIN || errno == EWOULDBLOCK){
                free(mess_ptr -> oggetto);
                free(mess_ptr);
                TIMEOUT_OCCURRED(i);
                }else if(errno == EINTR){
                        continue;
                }else{
                        free(mess_ptr -> oggetto);
                        free(mess_ptr);
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                }
        }
if((mess_ptr -> testo = malloc(size)) == NULL){
```

```
        printf("malloc failed, errno: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    while(recv(sock_des[i], mess_ptr -> testo, size, 0) == -1){
            if(errno == EAGAIN || errno == EWOULDBLOCK){
                    free(mess_ptr -> oggetto);
                    free(mess_ptr -> testo);
                    free(mess_ptr);
                    TIMEOUT_OCCURRED(i);
                    }else if(errno == EINTR){
                            continue;
                    }else{
                            free(mess_ptr -> oggetto);
                            free(mess_ptr -> testo);
                            free(mess_ptr);
                            printf("recv failed, errno: %s\n", strerror(errno));
                            exit(EXIT_FAILURE);
                    }
            }

            mess_ptr -> next = NULL;//essendo l'ultimo elemento della lista non punta a nulla (non è
una lista circolare)

    pthread_mutex_lock(&mutex_mess);
    inserisci_mess_in_lista(mess_ptr, index);
    pthread_mutex_unlock(&mutex_mess);
}



void leggi_mess(int i){
    char *buffer;
    struct messaggio *curr;
    int size;

    pthread_mutex_lock(&mutex_mess);
    if(head_list[i].head == NULL){//lista vuota
            size = strlen("\n\nnon ci sono messaggi da leggere\n\n") + 1;
            while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
        while(send(sock_des[i], "\n\nnon ci sono messaggi da leggere\n\n", size, 0) == -1   &&   errno
== EINTR);
    }else{
            size = strlen("\n\nmessaggi a te inviati:\n") + 1;
            while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
        while(send(sock_des[i], "\n\nmessaggi a te inviati:\n", size, 0) == -1   &&   errno == EINTR);
        curr = head_list[i].head;
        while(curr != NULL){//si scorre tutta la lista per stamparne i contenuti
            size = strlen(curr -> mittente) + strlen(curr -> oggetto) + strlen(curr -> testo) +
strlen("\nMITTENTE:\n\t\nOGGETTO:\n\t\nTESTO:\n\t\n") + 1;
            while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
            if((buffer = malloc(size)) == NULL){
```

```
            printf("malloc failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
          }
          sprintf(buffer, "\nMITTENTE:\n\t%s\nOGGETTO:\n\t%s\nTESTO:\n\t%s\n", curr -> mittente,
curr -> oggetto, curr -> testo);
          while(send(sock_des[i], buffer, size, 0) == -1   &&   errno == EINTR);
          free(buffer);
          curr = curr -> next;
       }
       size = -1;
            while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
   }
   pthread_mutex_unlock(&mutex_mess);
}



void cancella_mess(int i){
   struct messaggio *next;
         int size;

   pthread_mutex_lock(&mutex_mess);
   if(head_list[i].head != NULL){
         head_list[i].tail = NULL;
      while(head_list[i].head != NULL){
         next = head_list[i].head -> next;
         free(head_list[i].head);
         head_list[i].head = next;
      }
   }
   size = strlen("fatto") + 1;
   while(send(sock_des[i], &size, sizeof(int), 0) == -1   &&   errno == EINTR);
   while(send(sock_des[i], "fatto", strlen("fatto") + 1, 0) == -1   &&   errno == EINTR);
   pthread_mutex_unlock(&mutex_mess);
}



void termina_connessione(int i){
         //cancellazione username e password dal file passwd
   char *content_file;
   int file_pointer = cerca_username_in_file(passwd, username[i]);
   int size;

   pthread_mutex_lock(&mutex_file);
   fseek(passwd, 0, SEEK_END);
   size = ftell(passwd) - (strlen(username[i]) + 64 + 1/*carattere '\n'*/);//taglia del file meno i bytes
della riga da eliminare

   if(size == 0){//basta troncare il file
      if((passwd = fopen("passwd", "w+")) == NULL){
```

```
            printf("fopen failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }else{//occorre ricopiare tutte le righe del file tranne quella da eliminare
        if((content_file = malloc(size + 1)) == NULL){
            printf("malloc failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
        fseek(passwd, 0, SEEK_SET);
                fread(content_file, 1, file_pointer, passwd);//scrittura sul buffer del file fino a
username da eliminare
        fseek(passwd, strlen(username[i]) + 65, SEEK_CUR);
        fread(&(content_file[file_pointer]), 1, size - file_pointer, passwd);//scrittura sul buffer del file
dalla riga successiva a quella da eliminare fino a EOF

                //file troncato
        if( (passwd = fopen("passwd","w+")) == NULL){
            printf("fopen failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
        //copia del buffer sul file troncato
        fprintf(passwd, "%s", content_file);
        fflush(passwd);
        free(content_file);
    }
    pthread_mutex_unlock(&mutex_file);
    cancella_mess(i);
    strcpy(username[i], "");

    while(send(sock_des[i], &size, sizeof(int), MSG_NOSIGNAL) == -1   &&   errno == EINTR);
//MSG_NOSIGNAL: flag per evitare SIGPIPE. Importante per evitare loop in caso di invocazione
da parte di handler a seguito di un SIGPIPE in un altro punto del codice
    sock_des[i] = -1;
    close(sock_des[i]);
}



void TIMEOUT_OCCURRED(int x){
            long fpointer;
            int check;

            pthread_mutex_lock(&mutex_file);
            fpointer = ftell(passwd);
            pthread_mutex_unlock(&mutex_file);
            if(fpointer != 0){
                    termina_connessione(x);
            }else{
                    sock_des[x] = -1;
                    close(sock_des[x]);
            }
```

```
        printf("\nthread %ld terminato\n", tid[x]);
        tid[x] = 0;
        max_clients++;
        pthread_exit(NULL);
}



void sigpipe_handler(int signo){
        pthread_t thread = pthread_self();
        int i = 0;

        puts("SIGPIPE RICEVUTA");
        while(thread != tid[i]) i++;//ciclo per trovare il tid del thread che ha ricevuto SIGPIPE
        termina_connessione(i);
        printf("\nthread %ld terminato\n", thread);
        tid[i] = 0;
        max_clients++;
        pthread_exit(NULL);
}



void handler(int signum){
   int i, check;
   puts("SEGNALE RICEVUTO");

        //ciclo per inviare SIGUSR1 a tutti i client connessi al server e cancellarne i messaggi
conservati nel server
        for(i=0; i<MAX_CLIENT; i++){
      if(tid[i] != 0){
        cancella_mess(i);
        }
        }
        fclose(passwd);
        exit(0);
}



void *thread(void *arg){
   int me = (int)arg;
   int size, choice = 0;
   char *password, *encrypted_password;
printf("\n___%d___%ld___\n", me, tid[me]);
   //ricezione username
   while(recv(sock_des[me], username[me], 129, 0) == -1){
        if(errno == EAGAIN || errno == EWOULDBLOCK){
                        TIMEOUT_OCCURRED(me);
                }else if(errno == EINTR){
                        continue;
```

```
                }else{
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                }
        }
    if(cerca_username_in_file(passwd, username[me]) != -1){
        while(send(sock_des[me], "utente già esistente, riprovare con username diverso dai seguenti
usernames:\n", strlen("utente già esistente, riprovare con username diverso dai seguenti
usernames:\n") + 1, 0) == -1   &&   errno == EINTR);
        print_users(me);
    }
    else{
        while(send(sock_des[me], "username utilizzabile", strlen("username utilizzabile") + 1, 0) ==
-1   &&   errno == EINTR);
        }

    //ricezione password
    while(recv(sock_des[me], &size, sizeof(int), 0) == -1){
        if(errno == EAGAIN || errno == EWOULDBLOCK){
                TIMEOUT_OCCURRED(me);
                }else if(errno == EINTR){
                        continue;
                }else{
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                }
        }
    if((password = malloc(size)) == NULL){
        printf("malloc failed, errno: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    while(recv(sock_des[me], password, size, 0) == -1){
        if(errno == EAGAIN || errno == EWOULDBLOCK){
                        free(password);
                        TIMEOUT_OCCURRED(me);
                }else if(errno == EINTR){
                        continue;
                }else{
                        free(password);
                        printf("recv failed, errno: %s\n", strerror(errno));
                        exit(EXIT_FAILURE);
                }
        }

    //criptazione password e inserimento nel file passwd
    encrypted_password = crypt(password, "$5$hakunamatataraga");
    pthread_mutex_lock(&mutex_file);
    fprintf(passwd, "%s:%s\n", username[me], encrypted_password);
    fflush(passwd);
    pthread_mutex_unlock(&mutex_file);
    free(password);
```

```
    while(1){
        while(recv(sock_des[me], &choice, sizeof(int), 0) == -1){
                        if(errno == EAGAIN || errno == EWOULDBLOCK){
                                TIMEOUT_OCCURRED(me);
                        }else if(errno == EINTR){
                                continue;
                        }else{
                                printf("recv failed, errno: %s\n", strerror(errno));
                                exit(EXIT_FAILURE);
                        }
          }

        switch(choice){
            case 1:
                if(autenticazione(me)) leggi_mess(me);
                break;//else si torna alla richiesta "cosa vuoi fare?"
            case 2:
                if(autenticazione(me)) spedisci_mess(me);
                break;
            case 3:
                if(autenticazione(me)) cancella_mess(me);
                break;
            case 4:
                if(autenticazione(me)) print_users(me);
                break;
            case 5:
                if(autenticazione(me)){
                      termina_connessione(me);
                                          printf("\nthread %ld terminato\n", tid[me]);
                                          tid[me] = 0;
                                          max_clients++;
                      pthread_exit(NULL);
                      }
                break;
        }
    }
}



int main(int argc, char **argv){
    struct sockaddr_in server;
    struct sockaddr client;
    struct timeval timeout;
    int server_sd;
    int addrlen = sizeof(client), i=0, check;

    timeout.tv_sec = TIMEOUT;
    timeout.tv_usec = 0; //inizializzato a 0 per evitare undefined behaviour
```

```
    for(; i<MAX_CLIENT; i++) sock_des[i]=-1;

            //gestione segnali
            signal(SIGHUP, handler);
            signal(SIGINT, handler);
            signal(SIGQUIT, handler);
            signal(SIGTERM, handler);
            signal(SIGPIPE, sigpipe_handler);

    //apertura file passwd
    if( (passwd = fopen("passwd","w+")) == NULL){
        printf("fopen failed, errno: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    //inizializzazione liste di messaggi
    for(i=0; i<MAX_CLIENT; i++){
            head_list[i].head = NULL;
            head_list[i].tail = NULL;
            }

            //inizializzazione struct sockaddr_in
            server.sin_family = AF_INET;
            server.sin_port = htons(PORT);
            server.sin_addr.s_addr = htonl(INADDR_ANY);
            bzero(&(server.sin_zero), 8); //server.sin_zero contiene così tutti zeri

            while(1){
                    while(!max_clients);//se limite threads è raggiunto, il main rimane bloccato su questo
ciclo, altrimenti procede con l'istruzione successiva

            //creaz socket
            if( (server_sd = (socket(AF_INET, SOCK_STREAM, 0))) == -1){
            printf("socket failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
            }

            if(setsockopt(server_sd, SOL_SOCKET, SO_REUSEADDR, &(int){1}, sizeof(int)) == -1){
            printf("setsockopt failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
            }

            //assegnazione indirizzo al socket
            if(bind(server_sd, (struct sockaddr *)&server, sizeof(server)) == -1){
            printf("bind failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
            }

            //socket pronto a ricevere richieste di connessione
            if(listen(server_sd, PENDING) == -1){
```

```
		printf("listen failed, errno: %s\n", strerror(errno));
		exit(EXIT_FAILURE);
		}

		printf("\n\nserver sulla porta %d in stato listening\n\n", PORT);

		//il main thread continuerà ad accettare eventuali nuove connessioni, gli altri thread
eseguiranno i servizi dell'applicazione
		i = 0;
		while(1){
				if(max_clients > 0){
					if(sock_des[i] == -1){//nel caso in cui il server sia sovraccarico, non si
entra nell'if
						while((sock_des[i] = accept(server_sd, &client, &addrlen)) == -
1);
						if(setsockopt(sock_des[i], SOL_SOCKET, SO_RCVTIMEO,
&timeout, sizeof(timeout)) == -1){//imposta sul socket di connessione creato un timeout
							printf("setsockopt failed, errno: %s\n", strerror(errno));
							exit(EXIT_FAILURE);
						}
						max_clients--;
				pthread_create(&(tid[i]), NULL, thread, (void *)i);
						printf("\nthread %ld creato\n", tid[i]);
					}
				i = (i+1) % MAX_CLIENT;
				}else{
					close(server_sd);//in questo modo la connect() del client ritorna errno
== ECONNREFUSED
					break;
				}
			}
		}
}
```

client.c

```
#include "lib.h"

char username[129];
int client_sd;
struct termios orig_term_conf;
pid_t pid;
int sem_des;
struct sembuf oper;


int registrazione_utente(int sockfd){
    char *password, *password_check, user[129], buffer[129];
    int check, size, valid_user = 0;

    //acquisizione username
    do{
```

```
        printf("Inserisci username: ");
        if(fgets(username, 129, stdin) == NULL){
        printf("fgets failed, errno: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
        }
        if(strstr(username, ":") != NULL){
        puts("lo username non può contenere il carattere ':'\n");
        continue;
        }else if(strcmp(username, "\n") == 0){
                puts("lo username deve essere lungo almeno un carattere\n");
                continue;
                }
        size = strlen(username);
        username[size - 1] = '\0';
        while(send(client_sd, username, size + 1, MSG_NOSIGNAL) == -1){
                if(errno == ECONNRESET   ||   errno == EPIPE){
                        puts("\nil server è stato chiuso\n");
                                close(client_sd);
                        exit(0);
                        }else if(errno == EINTR){
                                continue;
                        }else{
                                printf("send failed, errno: %s\n", strerror(errno));
                                exit(EXIT_FAILURE);
                        }
                }
        while(recv(client_sd, buffer, 129, 0) == -1   &&   errno == EINTR);
        if(strcmp(buffer, "utente già esistente, riprovare con username diverso dai seguenti
usernames:\n") == 0){
        puts("utente già esistente, riprovare con username diverso dai seguenti usernames:\n");
        while(1){
                check = recv(client_sd, user, 129, 0);
                if(check > 0){
                        printf("\t%s\n", user);
                        memset(user, 0, 129);
                }else if(check == -1   &&   errno == EINTR){
                        continue;
                        }else{
                                break;
                                }
        }
        }
        else if(strcmp(buffer, "username utilizzabile") == 0) valid_user = 1;
   }while(!valid_user);

   //set impostazioni terminale
   no_echo_input(&orig_term_conf);

   //acquisizione password
   do{
      printf("Inserisci password: ");
```

```
        if(scanf("%ms", &password) == EOF){
            free(password);
            printf("scanf failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
        puts("");
        CLEAR_INPUT_BUFFER;
        printf("Re-inserisci password: ");
        if(scanf("%ms", &password_check) == EOF){
            free(password_check);
            printf("scanf failed, errno: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
        CLEAR_INPUT_BUFFER;
        if( (check = strcmp(password, password_check)) != 0){
            puts("\nPassword diverse. Riprova, per favore!\n");
        }
    }while(check);

    //reset impostazioni terminale
    reset_echo_input(&orig_term_conf);

    //invio a server password
    size = strlen(password) + 1;
    while(send(sockfd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
    while(send(sockfd, password, size, 0) == -1   &&   errno == EINTR);

    free(password);
    free(password_check);
            puts("");
}



int autenticazione(){
    char *password;
    long file_pointer;
    int res, size;

    no_echo_input(&orig_term_conf);

            //acquisizione password dell'utente
    printf("password di %s: ", username);
    if(scanf("%ms", &password) == EOF){
        free(password);
        printf("scanf failed, errno: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
            CLEAR_INPUT_BUFFER;

    reset_echo_input(&orig_term_conf);
```

```
        size = strlen(password) + 1;
    while(send(client_sd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
    while(send(client_sd, password, size, 0) == -1   &&   errno == EINTR);

    while(recv(client_sd, &res, sizeof(int), 0) == -1   &&   errno == EINTR);
    puts("");
    return res;
}



void stampa_utenti(){
        int size, check;
        char *buffer;

        puts("\tlista degli utenti connessi al server:");
        while(1){
                check = recv(client_sd, &size, sizeof(int), 0);
                if(check == -1   &&   errno == EINTR) continue;
                else if(check > 0){
                        if(size == -1) break;
                        else{
                                if((buffer = malloc(size)) == NULL){
                                        printf("malloc failed, errno: %s\n", strerror(errno));
                                        exit(EXIT_FAILURE);
                                }
                                while(recv(client_sd, buffer, size, 0) == -1   &&   errno == EINTR);
                                printf("\t\t%s\n", buffer);
                                free(buffer);
                        }
                }else break;
        }
}



void read_msg(){
        int size, received_bytes;
        char *check, *buffer;

        while(recv(client_sd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
        if((check = malloc(size)) == NULL){
                printf("malloc failed, errno: %s\n", strerror(errno));
                exit(EXIT_FAILURE);
        }
    while(recv(client_sd, check, size, 0) == -1   &&   errno == EINTR);
    if(strcmp(check, "\n\nmessaggi a te inviati:\n") == 0){
        while(1){
                received_bytes = recv(client_sd, &size, sizeof(int), 0);
                if(received_bytes == -1   &&   errno == EINTR) continue;
```

```
                else if(received_bytes > 0){
                        if(size == -1) break;
                        else{
                                if((buffer = malloc(size)) == NULL){
                                        printf("malloc failed, errno: %s\n", strerror(errno));
                                        exit(EXIT_FAILURE);
                                }
                                recv(client_sd, buffer, size, 0);
                                printf("%s", buffer);
                                free(buffer);
                        }
                        }else break;
                }
    }else{
          printf("\n%s\n", check);
          }
    free(check);
    puts("");
}



void write_msg(){
          int size, found_dest = 0;
          char user[129], *check, *line = NULL;
          size_t len = 0;

          //acquisizione destinatario
          do{
                stampa_utenti();
                printf("inserisci username dell'utente cui vuoi spedire un messaggio: ");
        if(fgets(user, 129, stdin) == NULL){
           printf("fgets failed, errno: %s\n", strerror(errno));
           exit(EXIT_FAILURE);
        }
        user[strlen(user) - 1] = '\0';
        puts("");
        while(send(client_sd, user, 129, 0) == -1   &&   errno == EINTR);
        while(recv(client_sd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
        if((check = malloc(size)) == NULL){
           printf("malloc failed, errno: %s\n", strerror(errno));
           exit(EXIT_FAILURE);
           }
        while(recv(client_sd, check, size, 0) == -1   &&   errno == EINTR);
        if(strcmp(check, "username trovato") == 0){
           found_dest = 1;
           free(check);
           }
                else if(strcmp(check, "username non trovato") == 0){
                        puts("username non trovato");
                        free(check);
```

```
                }
        }while(!found_dest);

        //acquisizione oggetto del messaggio
        puts("inserisci oggetto del messaggio:");
   if(getline(&line, &len, stdin) == -1){
      printf("getline failed, errno: %s\n", strerror(errno));
      exit(EXIT_FAILURE);
   }
   puts("\n");
   size = strlen(line);
        line[size - 1] = '\0';
        while(send(client_sd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
   while(send(client_sd, line, size, 0) == -1   &&   errno == EINTR);

        //acquisizione testo del messaggio
   puts("inserisci testo del messaggio:");
   if(getline(&line, &len, stdin) == -1){
      printf("getline failed, errno: %s\n", strerror(errno));
      exit(EXIT_FAILURE);
   }
   puts("\n");
   size = strlen(line);
   line[size - 1] = '\0';
   while(send(client_sd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
   while(send(client_sd, line, size, 0) == -1   &&   errno == EINTR);
   free(line);
}



void del_msg(){
        int size;
        char *check;

        while(recv(client_sd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
        if((check = malloc(size)) == NULL){
                printf("malloc failed, errno: %s\n", strerror(errno));
                exit(EXIT_FAILURE);
        }
        while(recv(client_sd, check, size, 0) == -1   &&   errno == EINTR);
        puts("\n\nmessaggi cancellati con successo\n\n");
}



void close_client(){
        int size;

        while(recv(client_sd, &size, sizeof(int), 0) == -1   &&   errno == EINTR);
        puts("");
```

```
    close(client_sd);
    exit(EXIT_SUCCESS);
}



void handler(int signum){
        puts("\nSIGPIPE ricevuta, la connessione con il server è terminata inaspettatamente\n");
        reset_echo_input(&orig_term_conf); //in questo modo se il client viene interrotto mentre la
configurazione del terminale è modificata, verrà resettata
        close(client_sd);
        exit(0);
}



int main(int argc, char **argv){
    struct sockaddr_in server;
    int size = 0;
    long choice;
    char ch, str_choice[11], *endptr, user[129], *buffer, *check;
        char *line = NULL;
        size_t len = 0;

        //salva le impostazioni standard del terminale (serve in caso di chiamata agli handler dato
che loro non inizializzano la variabile orig_term_conf)
        tcgetattr(STDIN_FILENO, &orig_term_conf);

    //creazione socket
    if( (client_sd = (socket(AF_INET, SOCK_STREAM, 0))) == -1){
       printf("socket failed, errno: %s\n", strerror(errno));
       exit(EXIT_FAILURE);
    }

    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);

    //connessione del client al server
    while(connect(client_sd, (struct sockaddr *)&server, sizeof(server)) == -1){
        if(errno == EINTR){
                continue;
                }else if(errno == ECONNREFUSED){
                        printf("\nserver sovraccarico. Premere R per riprovare, qualsiasi altro
carattere per terminare\n");
                        ch = getchar();
                        CLEAR_INPUT_BUFFER;
                        if(ch == 82   ||   ch == 114) continue;
                        else exit(0);
                }else{
        printf("connect failed, errno: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
```

```
        }
    }
    puts("\nconnessione stabilita con successo");

    //gestione segnali (avviene dopo la connect() in modo tale che, se un client che si vuole connettere va in attesa che si liberi un posto nel server, può decidere di uscire dallo stato di attesa. In altre parole può interrompere il processo)
        signal(SIGINT, SIG_IGN);
        signal(SIGTERM, SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
        signal(SIGPIPE, handler);

    registrazione_utente(client_sd);

   //acquisizione input utente
    while(1){
        printf("\ncosa vuoi fare? " CORSIVO "(rispondi con numero corrispondente)" RESET "\n1. leggi messaggi per te\n2. spedisci messaggio\n3. cancella messaggi per te\n4. stampa utenti connessi al server\n5. termina connessione col server\n");
        if(fgets(str_choice, 11, stdin) == NULL){
            puts("fgets failed");
            exit(EXIT_FAILURE);
            }
                    str_choice[strlen(str_choice) - 1] = '\0';
        choice = strtol(str_choice, &endptr, 10);
        if(str_choice == endptr){
            puts("input invalido (1), riprova");
            continue;
        }else if(*endptr != '\0'){
            puts("input invalido (2), riprova");
            continue;
        }else if(choice < 1 || choice > 5){
            puts("input invalido (3), riprova");
            continue;
        }else{
            while(send(client_sd, &choice, sizeof(int), 0) == -1    &&    errno == EINTR);
        }

        switch(choice){
            case 1:
                if(autenticazione()){
                                                read_msg();
                }
                else puts("\npassword errata\n");
                                        break;
            case 2:
                if(autenticazione()){
                        while(send(client_sd, username, 129, 0) == -1    &&    errno == EINTR);
                    write_msg();
                }
                else puts("\npassword errata\n");
```

```
            break;
        case 3:
            if(autenticazione()){
                                        del_msg();
            }
            else puts("\npassword errata\n");
            break;
        case 4:
            if(autenticazione()){
                stampa_utenti();
            }
            else puts("\npassword errata\n");
            break;
        case 5:
            if(autenticazione()){
                close_client();
            }
            else puts("\npassword errata\n");
            break;
    }
  }
}
```