

# Informazioni

Data consegna progetto: 03/07/2024

Autori: Miranda Bellezza 7109803  
Daniele Fallaci 7112588  
Email di ateneo: [miranda.bellezza@edu.unifi.it](mailto:miranda.bellezza@edu.unifi.it)  
[daniele.fallaci@edu.unifi.it](mailto:daniele.fallaci@edu.unifi.it)

Sistemi Operativi utilizzati: Ubuntu 24.04 e EndeavourOS (arch-based)

Shell usate: bash e zsh

# Indice

<b>Informazioni</b> .....	1
<b>Indice</b> .....	1
<b>Descrizione Progetto</b> .....	3
Funzionalità.....	3
Struttura file.....	3
<b>Client</b> .....	4
Header .....	4
Main .....	5
Apertura socket e Connessione al server .....	5
Controllo argomenti.....	6
Lettura della risposta del Server .....	6
Gestione Input dell'utente .....	7
Funzioni.....	7
requestHome.....	7
parser .....	8
login.....	8
clean_stdin .....	9
<b>Server</b> .....	10
Libreria cJSON.....	10
Settings .....	10
Header .....	10
Main .....	11
Settings .....	11
Socket .....	12
Accept e Fork .....	12
Richiesta del client .....	13
Funzioni.....	13
createSettings .....	13
choiseHandler .....	13

sendMenu .....	15
loadDatabase .....	15
saveDatabase .....	16
readContent .....	16
printContent.....	17
search .....	17
removeFromList .....	19
editFromList .....	19
aggiungiPersona .....	20
creaPersona .....	21
login.....	22
Gestione Hashing.....	23
inToSha256.....	23
Altre funzioni in server_utils .....	24
clean_stdin .....	24
utils_strIncludeOnly.....	24
utils_isValidEmail .....	25
utils_sortByKey .....	25
<b>Signal Handler.....</b>	<b>26</b>
SIGINT .....	26
SIGPIPE.....	26
<b>Gestione comportamenti anomali .....</b>	<b>27</b>
Server.....	27
Arresto improvviso .....	27
Client.....	27
Inserimento troppi caratteri nell'input da tastiera .....	27
Inserimento linee di troppo durante l'attesa al semaforo.....	28
Arresto del client in sezione critica.....	29

## Descrizione Progetto

Il progetto consiste nella realizzazione di una rubrica telefonica in C, tramite socket del dominio AF INET, che opera secondo un'architettura client-server. La rubrica permette agli utenti di eseguire operazioni di gestione dei contatti, come cercare contatti, aggiungere, modificare e eliminare.

Il server prevede delle funzionalità per ogni utente (consultazione della rubrica e ricerca contatti) e delle operazioni che possono essere eseguite solo da un admin.

Le operazioni di modifica (modifica, aggiunta, rimozione) possono essere svolte solo da un admin per volta. Se qualcuno sta già modificando i contatti, i client successivi verranno messi in attesa (la sezione critica viene gestita tramite un semaforo).

Per evitare che il file del database diventi troppo pesante è stata data la possibilità di limitare il numero dei contatti, al momento il limite impostato è 100. E' possibile cambiarlo modificando `RECORDS_MAX` in "server.h".

Data la complessità dei metodi main, questi sono descritti in questa relazione con l'uso di pseudo-diagrammi di flusso, che non sono completi dal punto di vista algoritmico ma che permettono di capire il funzionamento logico.

## Funzionalità

- Consultazione completa della rubrica;
- Ricerca in rubrica;
- Procedura di login;
- Aggiunta contatti (solo previo login);
- Modifica contatti (solo previo login);
- Eliminazione contatti (solo previo login).

## Struttura file

I file del progetto sono suddivisi in due cartelle server e client che contengono rispettivamente i file del server e del client.

Entrambe le cartelle hanno un Makefile per compilare automaticamente i file.

Utilizzando il comando `make` vengono create le cartelle `../build`, che contiene i file `*.o`, e `../bin` che contiene i file compilati.

I Makefile possono eseguire il comando `make clean` per eliminare i file `*.o` e `make run` per far partire direttamente i programmi.

In più il Makefile del server può eseguire il comando `make deps` per installare le librerie mancanti, individuando il package manager di riferimento analizzando il file `"/etc/os-release"`.

All'interno della cartella vendor è stata inserita la libreria cJSON utilizzata dal server, in modo che sia disponibile e che non sia necessario installarla.

# Client

Il client fornisce un'interfaccia utente per interagire con la rubrica. Invia le richieste al server e visualizza i risultati.

Il programma del client può essere fatto partire in due modi:

```
./client -a <username> <password>
```

Aggiungendo lo username e la password come parametri è possibile fare il login direttamente sul server, mentre senza parametri

```
./client
```

o con username e password errati si entrerà come visitatori.

## Header

Sono stati creati due nuovi tipi di dato utilizzando `typedef` in combinazione con `struct` per rappresentare il messaggio `MSG` che viene scambiato tra client e server e le credenziali per il login.

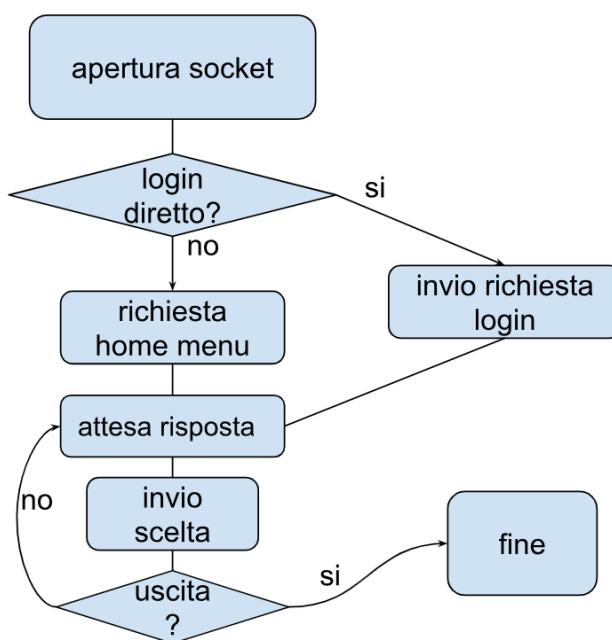
Il tipo `MSG` comprende un intero `isAdmin` e una stringa `message`.

Il tipo `credenziali` invece comprende due stringhe per contenere `username (user)` e `password (password)`.

```
// messaggio che client e server si scambiano           // credenziali per il login
typedef struct MSG {                                     typedef struct credenziali
    int isAdmin;                                         {
    char message[BUFFER_MAX];                           char user[25];
} MSG;                                                 char password[25];
} t_credenziali;
```

## Main

Diagramma del funzionamento del client:



## Apertura socket e Connessione al server

```
// apertura socket
if ((clientSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Errore nella creazione socket()\n");
    exit(-1);
}

// preparazione dell'indirizzo del server
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(SERVERPORT);
serverAddress.sin_addr.s_addr = inet_addr(SERVERADDRESS);

// connessione socket al server
if ((connect(clientSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress))) < 0) {
    perror("Errore nella connessione socket.\n");
    exit(-1);
}

printf("Sei connesso a %s\n", SERVERADDRESS);
```

Viene aperto un socket di tipo `AF_INET` che utilizza un protocollo per lo scambio di messaggi di tipo `SOCK_STREAM`.

Una volta aperto il socket, il client prova a connettersi e se la connessione va a buon fine viene stampata una stringa per indicare l'avvenuta connessione. Se c'è un errore nella creazione del socket o nel collegamento al server viene visualizzato un errore personalizzato e il programma termina.

## Controllo argomenti

```
// controllo argomenti per login diretto
if (argc == 1)
    requestHome(clientSocket, buffer);
else if (argc >= 4 && parser(argv, argc) == 1)
    login(clientSocket, buffer);
else {
    printf("Per il login diretto utilizzare: -a username password\nAccesso come visitatore.\n");
    requestHome(clientSocket, buffer);
}
```

Il client controlla se è stato avviato con degli argomenti:

- se non ci sono argomenti si passa alla richiesta di homepage ([requestHome \(\)](#)) come visitatore;
- se ci sono almeno 3 argomenti (“`-a <username> <password>`”) `argc >= 4` (perché il primo argomento è il nome del file) questi vengono passati ad un parser ([parser \(\)](#)) che li controlla e se il ritorno è affermativo viene avviato il login ([login \(\)](#));
- altrimenti vuol dire che ci sono argomenti ma non sono formattati correttamente per l'avvio del login diretto, viene quindi stampato un messaggio che spiega come avviare il login diretto ma si procede comunque alla richiesta di homepage come visitatore ([requestHome \(\)](#)).

## Lettura della risposta del Server

```

while(1){
    // attende risposta dal server
    strcpy(buffer.message, "");
    if(recv(clientSocket, &buffer, sizeof(buffer), 0) < 0) {
        perror("Errore nella ricezione dei dati.\n");
        exit(-1);
    } else {
        if(strcmp(buffer.message, "x") == 0) {
            close(clientSocket);
            printf("Disconnesso da %s\n", SERVERADDRESS);
            exit(0);
        }
        else
            printf("\nServer >>\n%s", buffer.message);
    }
}

```

Il client entra in un ciclo while(1), pulisce il messaggio condiviso e prova a leggere la risposta del server attraverso una `recv()`.

Se c'è un errore nella lettura viene stampato un messaggio di errore e il client si disconnette.

Se c'è una risposta:

- se la risposta è “x”: il server sta comunicando al client di disconnettersi, viene chiuso il socket, stampato un messaggio per indicare la disconnessione e il programma termina,
- altrimenti viene stampato il messaggio ricevuto.

## Gestione Input dell'utente

```

// Controllo e pulizia stream input
clean_stdin();

// manda messaggio al server
printf(">>\t");
fgets(buffer.message, strlen(buffer.message), stdin);
printf("\n");

// se la stringa inviata è maggiore della dimensione permessa viene tagliata
if (strlen(buffer.message) > INPUT_MAX)
    buffer.message[INPUT_MAX - 1] = '\0';
else
    buffer.message[strlen(buffer.message) - 1] = '\0';

send(clientSocket, &buffer, sizeof(buffer), 0);
}

return(0);
}

```

L'utente ha la possibilità di scegliere l'operazione da effettuare attraverso un input da tastiera.

`stdin`, che contiene gli input da tastiera, viene ripulito (`clean_stdin()`) in modo da assicurarci che l'input sia sempre inserito dopo il messaggio del server.

Viene quindi richiamato `fgets()` che legge sul buffer una nuova riga.

Il contenuto del buffer appena inserito viene controllato. Se ha una dimensione maggiore di `INPUT_MAX` (definito nell'header) viene tagliato inserendo '\0' che indica la fine della

stringa, altrimenti se è più piccolo viene semplicemente sostituito l'ultimo carattere ('\n') con '\0'.

Infine, la stringa viene inviata al server attraverso una `send()`.

Ricomincia quindi il ciclo attendendo una nuova risposta dal server.

## Funzioni

### requestHome

```
static void requestHome(int clientSocket, MSG buffer) {
    strcpy(buffer.message, homeChar);
    send(clientSocket, &buffer, sizeof(buffer), 0);
}
```

La funzione `requestHome()` prende come argomenti il socket sul quale inviare il messaggio e un messaggio da inviare. Inserisce nel messaggio la richiesta dell'homepage (nel nostro caso "h") e la invia al server attraverso una `send()` sul socket.

### parser

```
// Gestione argomenti
static int parser(char *argomenti[]) {

    for (char **ptr = argomenti; ptr[2]; ptr++)
    {
        if(strcmp(ptr[0], loginArg) == 0){
            printf("Login richiesto con username: %s, password: %s", ptr[1], ptr[2]);
            strcpy(cred.user, ptr[1]);
            strcpy(cred.password, ptr[2]);
            return 1;
        }
    }
    return 0;
}
```

La funzione `parser()` prende come argomento i parametri passati da terminale.

Attraverso un ciclo for scorre un puntatore `ptr` ad ogni stringa finché il puntatore non punta alla terzultima stringa presente.

Se il puntatore punta a `loginArg` (per noi -a) il puntatore alle due stringhe successive viene copiato nelle credenziali (`cred.user` e `cred.password`).

## login

```
// Procedura login
static void login(int clientSocket, MSG buffer) {
    strcpy(buffer.message, loginChar);
    send(clientSocket, &buffer, sizeof(buffer), 0);

    strcpy(buffer.message, "");
    if(recv(clientSocket,&buffer, sizeof(buffer), 0) < 0)
    {
        perror("Errore nella ricezione dei dati.\n");
        exit(-1);
    }

    // invia il nome utente
    strcpy(buffer.message, cred.user);
    send(clientSocket,&buffer, sizeof(buffer), 0);

    strcpy(buffer.message, "");
    if(recv(clientSocket,&buffer, sizeof(buffer), 0) < 0)
    {
        perror("Errore nella ricezione dei dati.\n");
        exit(-1);
    }

    // invia la password
    strcpy(buffer.message,cred.password);
    send(clientSocket,&buffer, sizeof(buffer), 0);
}
```

La funzione `login()` segue la procedura manuale di [login](#) del server.

Copia nel buffer e invia la richiesta di login al server `loginChar` (nel nostro caso “l”), attende una risposta, copia nel buffer e invia il nome utente prendendolo dalle credenziali `cred.user`, attende la risposta del server e infine copia nel buffer e invia la password prendendola dalle credenziali `cred.password`.

## clean\_stdin

```
// Pulizia stream input
static int check_stdin()
{
    fd_set rfds;
    struct timeval tv;
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

    // tempo di attesa, 0 nel nostro caso
    tv.tv_sec = 0;
    tv.tv_usec = 0;

    return select(1, &rfds, NULL, NULL, &tv);
}

static void clean_stdin() {
    if(check_stdin())
    {
        char buffer[1024];

        while(check_stdin())
            // se ci sono righe piene le legge una alla volta
            fgets(buffer, sizeof(buffer), stdin);
    }
}
```

Le funzioni `check_stdin()` e `clear_stdin()` servono a controllare la presenza di dati all'interno dello standard input e ad eliminarli se presenti.

Sono utili per regolare la quantità di input da tastiera che l'utente può inviare.

La funzione `check_stdin()` richiama la funzione `select()` che ci permette di monitorare descrittori di file e qui in particolare viene usata per monitorare 0 (`stdin`).

Gli argomenti passati alla `select()` sono il numero di descrittori da monitorare (1), il descrittore da monitorare per la lettura (`rfds` settato su 0 con `FD_ZERO(&rfds)` e `FD_SET(0, &rfds)`), i descrittori da monitorare per la scrittura (`NULL`), i descrittori da monitorare per presenza di errori (`NULL`) e una struct di tipo `timeval` che indica per quanto tempo devono essere monitorati, nel nostro caso abbiamo bisogno di una sola valutazione quindi il tempo di attesa è 0.

La funzione `clean_stdin()` richiama `fgets()` che legge una riga per volta (quindi fino a che non trova "\n") finchè è presente qualcosa nello stream dello standard input controllando con `check_stdin()`.

## Server

Il server ha il compito di gestire la memorizzazione e la manipolazione dei dati dei contatti. Riceve le richieste dai client, esegue le operazioni richieste e invia i risultati.

Per la gestione dei dati utilizza due file:

- database.json: un file json che contiene i contatti salvati;
- setting.txt: che contiene i dati dell'amministratore.

## Libreria cJSON

La libreria cJSON consente un utilizzo di una notazione degli oggetti più sofisticata ed usata in numerose applicazioni moderne.

L'utilizzo della Javascript Object Notation ci permette di passare facilmente anche ad altri tipi di tecnologie come web-app, siti internet e database NoSQL(come mongoDB), garantendo una scalabilità del progetto per futuri ampliamenti.

È possibile visualizzare la documentazione qui:

<https://github.com/DaveGamble/cJSON>

## Settings

Le impostazioni del server sono composte semplicemente da un file settings.txt contenente le credenziali dell'amministratore.

Viene applicata una funzione hash alla password per proteggerla.

Se all'avvio non viene trovato il file delle impostazioni, si provvede alla creazione di questo e al salvataggio delle credenziali.

./server -r

E' possibile tramite il parametro `-r` resettare le credenziali.

## Header

Sono stati definiti tre nuovi tipi MSG, parallelo a quello in Client per consentire la comunicazione, t\_credenziali che è usato per salvare le credenziali e operationOnList usato per definire il tipo di operazione applicabile a seguito di una ricerca.

Inoltre sono definite alcune macro usate all'interno del codice.

```
#define SERVERPORT 12345
#define SERVERADDRESS "127.0.0.1"

#define BUFFER_MAX 1024
#define MAX_CLIENT 5
#define RECORDS_INPAGE 4
#define RECORDS_MAX 100

#define FILE_USERS "settings.txt"
#define FILE_DB "data.json"
#define CONVERSION_SHA256_MAX (EVP_MAX_MD_SIZE *2 + 1)
```

```

typedef struct MSG {
    int isAdmin;
    char message[BUFFER_MAX];
} MSG;

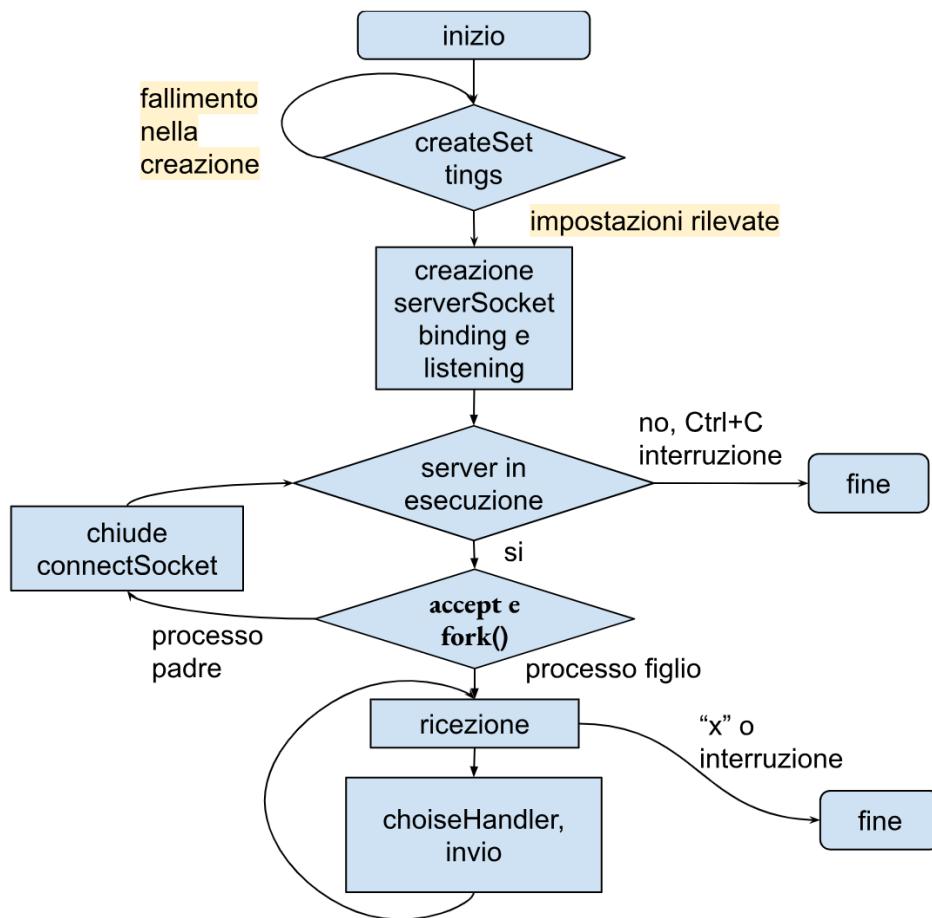
typedef int (*operationOnList)(cJSON *found, cJSON* list, int connectSocket, MSG buffer);

typedef struct credenziali
{
    char user[25];
    char password[25];
} t credenziali;

```

## Main

Diagramma del funzionamento del server:



## Settings

```

// Creazione setting
while(createSettings(argv,argc) != 1){
    printf("Setting:\n");
}

```

Viene richiamata la funzione [createSetting\(\)](#) che controlla se esiste un file setting.txt e lo ripristina in caso non esista o in caso si apra il server utilizzando il flag di reset -r.

## Socket

```
// creazione del socket
if((serverSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Errore nella creazione del socket.\n");
    exit(-1);
}

// gestione dei segnali
signal(SIGPIPE, customSigPipeHandler);
signal(SIGINT,customSigHandler);

serverAddress.sin_family = AF_INET; // IPv4
serverAddress.sin_port = htons(SERVERPORT); // porta
serverAddress.sin_addr.s_addr = inet_addr(SERVERADDRESS); // indirizzo server

// bind del socket
if((returnCode = bind(serverSocket,(struct sockaddr*) &serverAddress, sizeof(serverAddress)))< 0){
    perror("Errore nel binding del socket.\n");
    exit(-1);
}

// listen
if((returnCode = listen(serverSocket,MAX_CLIENT)) < 0){
    perror("Errore nella listen sul socket.\n");
    exit(-1);
}

printf("Server online CTRL+C per terminare\n");
```

Viene creato un socket di tipo `AF_INET` che utilizza un protocollo per lo scambio di messaggi di tipo `SOCK_STREAM`.

Viene assegnato un indirizzo al socket attraverso la funzione `bind()` e il socket si mette in ascolto con la funzione `listen()`.

Se tutto va a buon fine viene stampata una stringa per indicare che il server è pronto a connessioni dei client. Se invece si presenta un errore viene visualizzato un messaggio di errore personalizzato e il programma termina.

## Accept e Fork

```
while(1)
{
    if((connectSocket = accept(serverSocket, (struct sockaddr *)&clientAddress, &clientAddressLen)) < 0){
        perror("Errore in accept.\n");
        close(serverSocket);
        exit(-1);
    }
    // crea un processo figlio con fork()
    if(fork() == 0){
        clientIP = inet_ntoa(clientAddress.sin_addr);
        printf("Client connesso! @ %s : %d PID: %d PPID: %d\n",clientIP, connectSocket,getpid(),getppid());
        ...
    }
    else
        close(connectSocket);
}
```

Appena un client si connette al socket si ha una `fork()` del processo.

Il processo si divide in:

- processo padre: che chiude la connessione appena creata e attende una nuova connessione;
- processo figlio: che gestisce le richieste del client connesso.

## Richiesta del client

```

while(1) {
    strcpy(buffer.message, "");

    // attende richiesta dal client
    if((returnCode = recv(connectSocket, &buffer, sizeof(buffer), 0)) < 0) {
        printf("Client @ %s : %d - Errore nella ricezione dei dati.\nClient @ %s : %d disconnesso.\n",clientIP,connectSocket,clientIP,connectSocket);
        close(connectSocket);
        exit(-1);
    } else {
        printf("Client @ %s : %d - isAdmin: %d, Message %s\n",clientIP,connectSocket,buffer.isAdmin,buffer.message);

        if(strcmp(buffer.message, "x") == 0)
        {
            send(connectSocket, &buffer, sizeof(buffer), 0);
            printf("Client @ %s : %d disconnesso.\n", clientIP, connectSocket);
            close(connectSocket);
            // chiude il processo figlio
            exit(0);
        }

        // gestione delle richieste
        choiseHandler(connectSocket, clientIP, buffer.sem);
    }
}

```

Il processo figlio attende con una `recv()` il messaggio del client. Se c'è un errore chiude la connessione, altrimenti lo stampa.

Se il messaggio è uguale a “x” il client ha deciso di chiudere la connessione, il server risponde rimandando “x” e chiude la connessione e il processo.

Se invece si ha una risposta diversa questa viene passata alla funzione [choiseHandler\(\)](#) che la gestisce.

## Funzioni

### createSettings

Si occupa di creare le impostazioni del server. E' invocata al primo avvio e in modalità di reset col parametro `-r` da terminale.

Si occupa della scelta di user e password dell'amministratore e le immagazzina nel file di riferimento. Utilizza il metodo [intoSha256\(\)](#) per la protezione del server e un metodo `parser` per controllare la presenza del parametro di reset.

Fa due importanti accessi:

- Apertura in lettura del file
- apertura in scrittura e creazione in caso di assenza del file o richiesta di reset

Le impostazioni sono salvate in un file .txt per semplicità e necessità di salvare solo delle credenziali.

### choiseHandler

La funzione `choiseHandler()` legge il messaggio inviato dal client e attraverso uno switch richiama le funzioni richieste.

- “h” -> [sendMenu\(\);](#)
- “v” -> [readContent\(\);](#)
- “s” -> [search\(\);](#)
- “l” -> [login\(\);](#)
- “a” -> procedura per aggiunta;
- “m” -> procedura per modifica;
- “r” -> procedura per rimozione;

- default: invia “Comando non riconosciuto.”

Le procedure per aggiunta, modifica e rimozione sono strutturate in modo analogo.  
Ad esempio possiamo vedere l'aggiunta:

```
case 'a':
    if (buffer.isAdmin == 1){
        // controlla semaforo
        sem_getvalue(sem,&valueSem);

        // attesa se semaforo occupato
        if(valueSem == 0){
            strcpy(buffer.message,"Qualcun altro sta modificando i contatti. Vuoi attendere? [y/N]\n");
            send(connectSocket,&buffer, sizeof(buffer), 0);

            strcpy(buffer.message, "");
            if(recv(connectSocket,&buffer,sizeof(buffer), 0) < 0) {
                printf("Errore nella ricezione dei dati.\n");
                strcpy(buffer.message, "Errore nella ricezione dei dati.\n");
                break;
            }
            if(strcmp(utils_lowercase(buffer.message), "y") != 0) {
                strcpy(buffer.message, "");
                break;
            }
            printf("Client @ %s : %d in attesa per l'aggiunta di contatti.\n", clientIP, connectSocket);
        }
        // chiude semaforo
        sem_wait(sem);

        semPtr = &sem;
        inCriticalSection = getpid();
        printf("Client @ %s : %d sta aggiungendo contatti.\n", clientIP, connectSocket);

        int isAdded = aggiungiPersona(connectSocket, clientIP, buffer);

        // libera semaforo
        sem_post(sem);
        inCriticalSection = 0;
    }
```

Si controlla se il messaggio arriva da un admin. Si guarda il semaforo e se è occupato si chiede al client se vuole rimanere in attesa o tornare al menu.

Allo sblocco del semaforo, si richiama la funzione [aggiungiPersona\(\)](#) e si procede a liberare il semaforo.

La modifica e la rimozione procedono in modo analogo, richiamando però la funzione [search\(\)](#) con le rispettive operazioni [editFromList\(\)](#) e [removeFromList\(\)](#).

```
// torna direttamente al menu
if(strlen(buffer.message) < 42) { // se il buffer contiene una stringa singola es.("Attenzione username o password errati.")
    sendMenu(connectSocket, buffer);
    return 1;
}

// dopo aver eseguito correttamente l'operazione di visita o ricerca chiede se si vuole tornare al menu
strcat(buffer.message, "Vuoi tornare al menu? [y/N]\n");
send(connectSocket,&buffer, sizeof(buffer), 0);

strcpy(buffer.message, "");
if(recv(connectSocket,&buffer,sizeof(buffer), 0) < 0 || strcmp(utils_lowercase(buffer.message), "y") != 0) {
    strcpy(buffer.message, "x");
    send(connectSocket, &buffer, sizeof(buffer), 0);

    close(connectSocket);
    printf("\nClient con pid %d disconnesso.\n",getpid());
    exit(0);
}

strcpy(buffer.message, "");
sendMenu(connectSocket, buffer);
return 1;
}
```

Dopo la conclusione della procedura principale si torna al menù con [sendMenu\(\)](#) oppure dopo l'esecuzione di visita o ricerca si richiede al client se vuole continuare con il menù o uscire.

## sendMenu

```
static void sendMenu(int connectSocket, MSG buffer)
{
    // banner
    strcat(buffer.message, divisore);
    strcat(buffer.message, menuHeader);
    strcat(buffer.message, divisore);
    strcat(buffer.message, "\n");

    strcat(buffer.message, scelte);

    // menu da admin
    if (buffer.isAdmin == 1)
        strcat(buffer.message, scelteadmin);
    else
        strcat(buffer.message, scelteLogin);

    // scelta per uscire
    strcat(buffer.message, sceltaUscita);

    send(connectSocket,&buffer, sizeof(buffer), 0);
}
```

Inserisce nel buffer il banner grafico della rubrica e il menù con le scelte, diverso se si è admin o visitatore.

**Interfaccia del client:** a sinistra si può vedere il menù da visitatore, a destra il menù da admin.

<p>Rubrica Telefonica</p> <hr/> <p>v - visita s - ricerca l - login admin x - esci</p> <p>Cosa vuoi fare?</p> <p>&gt;&gt;</p>	<p>Rubrica Telefonica</p> <hr/> <p>v - visita s - ricerca a - aggiungi contatto m - modifica r - rimuovi contatto x - esci</p> <p>Cosa vuoi fare?</p> <p>&gt;&gt;</p>
---	---

## loadDatabase

```
static cJSON * loadDatabase()
{
    cJSON *jsonArray;

    // apre file data in lettura
    FILE *fp = fopen(FILE_DB, "r");
    if (fp == NULL) {
        // se il database non esiste
        printf("Errore in lettura database: database non trovato.\n");
        return NULL;
    } else {
        // calcolo dimensione del database
        fseek(fp, 0, SEEK_END);
        long size = ftell(fp);
        fseek(fp, 0, SEEK_SET);

        // memoria per database
        char *fileContent = calloc((size + 1), sizeof(char));
        fread(fileContent, sizeof(char), size, fp);

        fclose(fp);

        // parse della stringa json
        jsonArray = cJSON_Parse(fileContent);

        free(fileContent);
    }
    return jsonArray;
}
```

La funzione restituisce un array cJSON con il contenuto del file database.json (NULL se il file non viene trovato).

Viene aperto il file database.json.

Se non nullo, la dimensione del file viene calcolata utilizzando fseek() e ftell(). Con fseek() si punta alla fine del file, si legge con ftell() la posizione del puntatore e si torna all'inizio del file sempre con fseek().

Viene allocata una quantità di spazio necessaria per la lettura del file utilizzando calloc() e il file viene letto e chiuso. Il contenuto del file viene quindi sottoposto ad un parse cJSON\_Parse() (libreria cJSON) che crea l'array JSON.

## saveDatabase

```
static void saveDatabase(cJSON * jsonArray){
    // sort dell'array per cognome
    utils_sortByKey(jsonArray, "surname");
    // creazione di una stringa in formato json con tutti gli oggetti
    char *json_str = cJSON_Print(jsonArray);

    // salva la stringa JSON sul file database
    FILE * fp = fopen(FILE_DB, "w");
    if (fp == NULL) {
        printf("impossibile aprire il file dei dati\n");
        exit(-1);
    }
    fputs(json_str, fp);

    fflush(fp);
    fclose(fp);
    // free the JSON string and cJSON object
    cJSON_free(json_str);
}
```

La funzione salva sul file database.json il contenuto di un array di cJSON.

Gli elementi presenti nell'array vengono ordinati per chiave ("surname") e con cJSON\_Print() (libreria cJSON) viene creata la stringa corrispondente all'array.

Viene quindi aperto il file database.json e sovrascritto con la nuova stringa.

## readContent

```
static MSG readContent(int connectSocket, char*clientIP,MSG buffer)
{
    cJSON *jsonArray = loadDatabase();
    strcpy(buffer.message, "");
    return printContent(jsonArray,connectSocket,clientIP,buffer);
}
```

I contatti vengono caricati su un array cJSON attraverso [loadDatabase\(\)](#). Viene poi invocata [printContent\(\)](#).

## printContent

```
static MSG printContent(cJSON * array, int connectSocket, char*clientIP,MSG buffer)
{
    char elementString[1024];
    int nContacts = 0;
    int nPage = 1;

    if (array == NULL || cJSON_GetArraySize(array) == 0)
        strcat(buffer.message, "Non ci sono contatti salvati.\n");
    else
    {
        nContacts = cJSON_GetArraySize(array);

        cJSON *element = array->child;
        int i = 1;

        while (element)
        {
            // Codice per stampare l'elemento
            // ...
        }
    }
}
```

La funzione `printContent()` prende un array cJSON, crea una stringa per ogni elemento nell'array e li invia al client una “pagina” per volta. La variabile `i` indica l'indice del contatto +1, quindi il numero del contatto (partendo da 1).

La pagina viene costruita così:

```
while (element)
{
    // Inizio pagina
    if(i%RECORDS_INPAGE == 1) {
        strcpy(buffer.message, divisore);
        strcat(buffer.message, cercaHeader);
        strcat(buffer.message, divisore);
        snprintf(elementString, sizeof(elementString), "%d contatti trovati. Pagina %d di %d.\n", nContacts, nPage, ((nContacts+(RECORDS_INPAGE-1))/RECORDS_INPAGE));
        strcat(buffer.message, elementString);
    }

    // Stringa Contatto
    sprintf(elementString, sizeof(elementString), "%d) Nome: %s\n Cognome: %s\n Età: %d\n Email: %s\n Telefono: %s\n\n", i, cJSON_GetObjectItem(element, "name")->val);
    strcat(buffer.message, elementString);

    // Fine pagina
    if (i < nContacts && i%RECORDS_INPAGE == 0)
    {
        strcat(buffer.message, "Vuoi vedere la pagina successiva? [y/N]\n");
        send(connectSocket,&buffer, sizeof(buffer),0);

        strcpy(buffer.message, "");
        if((recv(connectSocket,&buffer,sizeof(buffer), 0)) < 0) {
            printf("Errore nella ricezione dei dati.\n");
            close(connectSocket);
            exit(-1);
        }

        printf("Client @ %s : %d - User: %s\n", clientIP, connectSocket, buffer.message);

        if (strcmpiutils.lowercase(buffer.message), "y") != 0) {
            strcpy(buffer.message, "");
            break;
        }
        nPage++;
        strcpy(buffer.message, "");
    }

    element = element->next;
    i++;
}
}

return buffer;
```

Per l'inizio della pagina viene inserita una stringa con il numero di contatti trovati e il numero di pagina, poi vengono inserite le stringhe dei contatti fino ad avere un numero di contatti per pagina prestabilito (`RECORDS_INPAGE = 4`).

Infine, se ci sono ancora contatti da stampare viene inserita nel buffer una stringa per chiedere al client se vuole visualizzare la pagina successiva.

Il tutto viene inviato al client sul socket.

## search

La funzione è in grado di individuare i contatti che hanno un determinato nome, cognome, nome cognome, cognome nome o una qualsiasi sottostringa di questi.

```

static MSG search(int connectSocket, char*clientIP,MSG buffer, operationOnList op)
{
    cJSON *jsonArray = loadDatabase();
    cJSON *foundArr = cJSON_CreateArray();
    char searchName[25];

    if (jsonArray == NULL || cJSON_GetArraySize(jsonArray) == 0) {
        strcpy(buffer.message, "Non ci sono contatti salvati.\n");
        return buffer;
    }

    strcpy(buffer.message, "Inserire nome della persona da cercare:\n");
    send(connectSocket,&buffer, sizeof(buffer),0);

    strcpy(buffer.message, "");
    if((recv(connectSocket,&buffer,sizeof(buffer), 0)) < 0) {
        printf("Client @ %s : %d - Errore nella ricezione dei dati.\nClient @ %s : %d disconnesso.\n",clientIP,connectSocket,clientIP,connectSocket);
        close(connectSocket);
        exit(-1);
    } else {
        strcpy(searchName, buffer.message);
        printf("Client @ %s : %d - Ricerca: %s\n", clientIP, connectSocket, buffer.message);
    }
}

```

I contatti vengono caricati sulla variabile `foundArray` con [loadDatabase\(\)](#). Poi viene richiesto al client un nome da ricercare e viene salvato nella variabile `searchName`.

```

cJSON *element = jsonArray->child;
char nameSurname[25];
char surnameName[25];

// cerca se il nome inserito è sottostringa di "nome cognome" o "cognome nome"
while (element)
{
    sprintf(nameSurname, sizeof(nameSurname), "%s %s", cJSON_GetObjectItem(element,"name")->valuestring, cJSON_GetObjectItem(element,"surname")->valuestring);
    sprintf(surnameName, sizeof(surnameName), "%s %s", cJSON_GetObjectItem(element,"surname")->valuestring, cJSON_GetObjectItem(element,"name")->valuestring);

    if(strstr(nameSurname, utils_lowercase(searchName)) || strstr(utils_lowercase(surnameName), utils_lowercase(searchName)))
        cJSON_AddItemToArray(foundArr, cJSON_Duplicate(element, i));

    element = element->next;
}

cJSON_Delete(element);

strcpy(buffer.message, "");

if(cJSON_GetArraySize(foundArr) > 0)
{
    buffer = printContent(foundArr,connectSocket,clientIP,buffer);

    if(op != NULL)
        strcpy(buffer.message, (op(foundArr,jsonArray,connectSocket,clientIP, buffer)? "Modifica dei contatti effettuata.\n" : "Nessuna modifica effettuata\n"));
}
else
    strcpy(buffer.message, "Nessun contatto con questo nome.\n");

return buffer;
}

```

Viene fatto un ciclo sugli elementi dell'array cJSON (`jsonArray`) e ricercato il nome inserito come sottostringa `strstr()` della stringa “nome cognome” o “cognome nome” dell’elemento dell’array.

Se viene trovata una sottostringa l’elemento viene aggiunto ad un nuovo array `foundArr`. Se quest’ultimo array non è vuoto viene invocata [printContent\(\)](#) per inviare il risultato della ricerca al client.

La funzione `search` supporta due operazioni i cui puntatori possono essere passati come argomento della funzione:

- [removeFromList](#)
- [editFromList](#)

Passando invece NULL la funzione termina qui (questo avviene nel caso della ricerca da visitatore).

## removeFromList

```

static int removeFromList(cJSON *found, cJSON* list,int connectSocket, char *clientIP, MSG buffer)
{
    int num = -1;
    int nContacts = cJSON_GetArraySize(found);

    do
    {
        strcpy(buffer.message,"Inserisci l'indice del contatto da eliminare. x per tornare al menu\n");
        send(connectSocket,&buffer, sizeof(buffer), 0);

        strcpy(buffer.message, "");
        if(recv(connectSocket,&buffer,sizeof(buffer), 0) < 0){
            printf("Errore nella ricezione dei dati.\n");
            return 0;
        }

        if (strcmp(utils_lowercase(buffer.message), "x") == 0)
            return 0;
        num = atoi(buffer.message);
        strcpy(buffer.message, "");

    } while(num < 1 || num > nContacts);

    // trova elemento corrispondente nell'elenco contatti
    int index = 0;
    cJSON * elementDeleted = cJSON_GetArrayItem(found, (num-1));
    cJSON * element = list->child;
    while(element){
        if(strcmp(cJSON_Print(elementDeleted), cJSON_Print(element)) == 0)
            break;
        index++;
        element= element->next;
    }

    char elementStr[1024];
    sprintf(elementStr, sizeof(elementStr), "Contatto:\nNome: %s\nCognome: %s\nEtà: %d\nEmail: %s\nATTENZIONE. Vuoi confermare la rimozione? [y/N]\n",
            cJSON_GetObjectItem(element,"name")->valuestring, cJSON_GetObjectItem(element,"surname")->valuestring,
            cJSON_GetObjectItem(element,"age")->valueint, cJSON_GetObjectItem(element,"email")->valuestring,
            cJSON_GetObjectItem(element,"phone")->valuestring);
    strcpy(buffer.message, elementStr);
    send(connectSocket,&buffer, sizeof(buffer), 0);

    if(recv(connectSocket,&buffer,sizeof(buffer), 0) < 0)
        printf("Errore nella ricezione dei dati.\n");
    return 0;
}

if(strcmp(utils_lowercase(buffer.message), "y") != 0)
    return 0;

printf("Client @ %s : %d - eliminazione di %s\n", clientIP, connectSocket,cJSON_Print(element));

//cancella dall'array all'indirizzo selezionato
cJSON_DeleteItemFromArray(list,index);
saveDatabase(list);
return 1;
}

```

La funzione `removeFromList()` continua la `search()`. Riprende da dopo il `printContent()` e al client viene richiesto di inserire l'indice del contatto da eliminare tra quelli che gli vengono proposti sullo schermo (oppure tornare al menù).

Il numero inviato dal client viene controllato (deve rientrare tra quelli proposti, altrimenti viene chiesto di nuovo) e successivamente il contatto corrispondente viene ricercato nella lista contatti.

Viene richiesto al client di confermare l'eliminazione e se la conferma è positiva il contatto viene eliminato con `cJSON_DeleteItemFromArray()` (libreria cJSON) e la funzione restituirà 1. Altrimenti se non c'è nessun contatto eliminato la funzione restituisce 0.

## editFromList

Ha una struttura simile alla `removeFromList()`. Dopo aver richiesto un indice valido e dopo la conferma di voler modificare viene avviata la funzione [`creaPersona\(\)`](#).

Se viene creato correttamente un nuovo contatto il precedente viene sostituito tramite `cJSON_ReplaceItemInArray()` (libreria cJSON).

```

char elementStr[1024];
snprintf(elementStr, sizeof(elementStr), "Contatto:\n\tNome: %s\n\tCognome: %s\n\tEtà: %d\n\tEmail: %s\n\tTelefono: %s\nATTENZIONE. Vuoi modificarlo? [y/N]\n",
         cJSON_GetObjectItem(element, "name")->valuestring, cJSON_GetObjectItem(element, "surname")->valuestring,
         cJSON_GetObjectItem(element, "age")->valueint, cJSON_GetObjectItem(element, "email")->valuestring,
         cJSON_GetObjectItem(element, "phone")->valuestring);
strcpy(buffer.message, elementStr);
send(connectSocket, &buffer, sizeof(buffer), 0);

if((recv(connectSocket, &buffer, sizeof(buffer), 0)) < 0) {
    printf("Errore nella ricezione dei dati.\n");
    return 0;
}
if(strcmp(utils_lowercase(buffer.message), "y") != 0)
    return 0;

//edit
printf("Client @ %s : %d - modifica di %s\n", clientIP, connectSocket, cJSON_Print(element));
cJSON* nuovaPersona = creaPersona(connectSocket, buffer);

if(nuovaPersona != NULL)
{
    printf("Client @ %s : %d - modificato con %s\n", clientIP, connectSocket, cJSON_Print(nuovaPersona));
    cJSON_ReplaceItemInArray(list, index, nuovaPersona);
    saveDatabase(list);

    cJSON_Delete(nuovaPersona);
    return 1;
}
printf("Client @ %s : %d - modifica annullata.\n", clientIP, connectSocket);
cJSON_Delete(nuovaPersona);
return 0;
}
    
```

## aggiungiPersona

```

static int aggiungiPersona(int connectSocket, char *clientIP, MSG buffer){
    cJSON *jsonArray = loadDatabase();

    if (jsonArray == NULL) {
        printf("Non sono presenti contatti. Creazione nuovo database.\n");
        jsonArray = cJSON_CreateArray();
    }

    if(cJSON_GetArraySize(jsonArray) > RECORDS_MAX)
        return -1;

    cJSON* jsonItem = creaPersona(connectSocket, buffer);
    if(jsonItem != NULL)
    {
        cJSON_AddItemToArray(jsonArray, jsonItem);
        printf("Client @ %s : %d ha inserito:\n%s\n", clientIP, connectSocket, cJSON_Print(jsonItem)); //stampa solo la persona appena inserita
        saveDatabase(jsonArray);
        cJSON_Delete(jsonItem);
        return 1;
    }
    cJSON_Delete(jsonItem);
    return 0;
}
    
```

La funzione `aggiungiPersona()` ritorna -1 se si è raggiunto il massimo dei contatti permessi, 1 se il nuovo contatto è stato aggiunto correttamente e 0 se il contatto non è stato aggiunto.

La funzione carica i contatti attraverso `loadDatabase()`. Se il file non esiste crea un nuovo array e se i contatti sono maggiori di `RECORD_MAX` la funzione ritorna -1;

Altrimenti invoca `creaPersona()` e se questa non è NULL la aggiunge all'array e salva il database con `saveDatabase()`.

## creaPersona

```

static cJSON *creaPersona(int connectSocket, MSG buffer)
{
    cJSON *jsonItem = cJSON_CreateObject();

    // banner
    strcpy(buffer.message, divisore);
    strcat(buffer.message, contattoHeader);
    strcat(buffer.message, divisore);

    // creazione del contatto
    strcat(buffer.message, "Inserire nome del contatto :\t");
    send(connectSocket,&buffer, sizeof(buffer),0);

    strcpy(buffer.message, "");
    if((recv(connectSocket,&buffer,sizeof(buffer), 0)) < 0) {
        printf("Errore nella ricezione dei dati.\n");
        return NULL;
    }
    else {
        while(strlen(buffer.message) < 1 || strlen(buffer.message) > 12) {
            strcpy(buffer.message, "Attenzione, non può essere lasciato vuoto e deve essere massimo 12 caratteri.\nInserire nome del contatto :\t");
            send(connectSocket,&buffer, sizeof(buffer),0);

            strcpy(buffer.message, "");
            if((recv(connectSocket,&buffer,sizeof(buffer), 0)) < 0) {
                printf("Errore nella ricezione dei dati.\n");
                return NULL;
            }
        }
        cJSON_AddStringToObject(jsonItem, "name", buffer.message);
    }
}

...
char elementStr[1024];
snprintf(elementStr, sizeof(elementStr), "Aggiungo:\n\tNome: %s\n\tCognome: %s\n\tEtà: %d\n\tEmail: %s\n\tTelefono: %s\nVuoi confermare? [y/N]\n",
         cJSON_GetObjectItem(jsonItem,"name")->valuestring, cJSON_GetObjectItem(jsonItem,"surname")->valuestring,
         cJSON_GetObjectItem(jsonItem,"age")->valueint, cJSON_GetObjectItem(jsonItem,"email")->valuestring,
         cJSON_GetObjectItem(jsonItem,"phone")->valuestring);
strcpy(buffer.message, elementStr);
send(connectSocket,&buffer, sizeof(buffer), 0);

strcpy(buffer.message, "");
if((recv(connectSocket,&buffer,sizeof(buffer), 0)) < 0) {
    printf("Errore nella ricezione dei dati.\n");
    return NULL;
} else if(strcmp(utils_lowercase(buffer.message), "y") != 0) {
    return NULL;
}
return jsonItem;
}

```

La funzione `creaPersona()` costruisce l'elemento cJSON contenente i dati di un contatto e lo restituisce. Se ci sono errori restituisce il valore NULL.

Per ogni campo del contatto (nome, cognome, età, email e telefono) viene richiesto al client di inviare il dato.

Al messaggio mandato dal client vengono applicati i seguenti controlli:

- il nome deve essere non nullo e massimo 12 caratteri;
- il cognome deve avere massimo 12 caratteri;
- l'età deve essere un numero compreso tra 1 e 100;
- l'email può essere lasciata vuota e se inserita deve avere massimo 30 caratteri ed essere un'email valida [`utils isValidEmail\(\)`](#);
- il numero di telefono può essere lasciato vuoto e se inserito deve avere massimo 16 caratteri ed essere formato solo dai caratteri "+ 1234567890" funzione [`utils strIncludeOnly\(\)`](#).

Dopo la creazione del contatto viene richiesta al client la conferma.

## login

```
static MSG login(int connectSocket, char *clientIP, MSG buffer){
    if(buffer.isAdmin == 1) {
        strcpy(buffer.message, "Sei già loggato.\n");
        return buffer;
    }
    t_credenziali cred;

    // Richiede User
    strcpy(buffer.message, "Inserire user:      ");
    send(connectSocket,&buffer, sizeof(buffer),0);

    strcpy(buffer.message, "");
    if((recv(connectSocket,&buffer,sizeof(buffer), 0)) < 0) {
        printf("Errore nella ricezione dei dati.\n");
        exit(-1);
    } else {
        strcpy(cred.user, buffer.message);
        printf("Client @ %s : %d - User: %s\n", clientIP, connectSocket, buffer.message);
    }

    // Richiede Password
    strcpy(buffer.message, "Inserire password:  ");
    send(connectSocket,&buffer, sizeof(buffer),0);

    strcpy(buffer.message, "");
    if((recv(connectSocket,&buffer,sizeof(buffer), 0)) < 0) {
        printf("Errore nella ricezione dei dati.\n");
        exit(-1);
    } else {
        strcpy(cred.password, buffer.message);
        printf("Client @ %s : %d - Password: %s\n", clientIP, connectSocket, buffer.message);
    }

    // verifica user e password
    if (verifica(cred)) {
        buffer.isAdmin = 1;
        strcpy(buffer.message, "Login effettuato con successo.\n\n");
        printf("Client @ %s : %d - Login effettuato.\n", clientIP, connectSocket);
    } else {
        strcpy(buffer.message, "Attenzione username o password errati.\n\n");
        printf("Client @ %s : %d - Username o password errati.\n", clientIP, connectSocket);
    }
}

return buffer;
```

La funzione `login()` richiede le credenziali al client e le salva su una variabile.

Il login utilizza una funzione di `verifica()` che apre il file `settings.txt` definito nell'header, effettua l'Hash della password e lo confronta con le credenziali fornite.

Utilizza [inToSha256\(\)](#) per la conversione.

```
static int verifica(t_credenziali cred)
{
    FILE * fptr;
    t_credenziali admin;
    int login = 0;

    fptr = fopen(FILE_USERS, "r");
    if (fptr == NULL) {
        printf("Errore file password non trovato.");
        return 0;
    } else {
        char hashPSWadmin[CONVENTION_SHA256_MAX];
        while(fscanf(fptr,"%s\n", admin.user, hashPSWadmin) != -1) {

            char hashPSWinserita[CONVENTION_SHA256_MAX];
            inToSha256(cred.password,hashPSWinserita);

            if ((strcmp(cred.user, admin.user) == 0) && (strcmp(hashPSWadmin, hashPSWinserita) == 0)) {
                login = 1;
                break;
            }
        }
    }
    fclose(fptr);
    return login;
}
```

## Gestione Hashing

La tecnica hashing è usata spesso nel contesto client-server o più in generale nei database. Si basa su una funzione irreversibile che permette di non salvare in chiaro le credenziali di utenti o informazioni sensibili generiche.

Esistono molti algoritmi di hash come MD5, SHA o SHA256, nel nostro contesto usiamo l'ultimo per salvare la password dell'admin.

### inToSha256

La funzione utilizza la libreria openssl, la quale gestisce molti tipi di algoritmi hash.

Viene creato un contesto di uso della conversione con EVP\_MD\_CTX\_NEW.

A questo punto si inizializza con EVP\_DigestInit\_ex, si aggiorna il contesto aggiungendo la stringa da convertire in EVP\_DigestUpdate e la conversione vera e propria avviene tramite EVP\_DigestFinal\_ex. In caso di errori o comportamenti anomali si ha una funzione di gestione degli errori (handleErrors).

```
void inToSha256(const char *inToHash, char *destination)
{
    unsigned char mid[EVP_MAX_MD_SIZE];
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
    if(mdctx == NULL) handleErrors();

    if(1 != EVP_DigestInit_ex(mdctx, EVP_sha256(), NULL))
        handleErrors();

    if(1 != EVP_DigestUpdate(mdctx, inToHash, strlen(inToHash)))
        handleErrors();

    unsigned int len = 0;
    if(1 != EVP_DigestFinal_ex(mdctx, mid, &len))
        handleErrors();

    EVP_MD_CTX_free(mdctx);

    hashToHexString(mid,len,destination);
}
```

Le funzioni di openssl non usano char, ma unsigned char come destinazione della conversione, quindi dobbiamo convertirla in un formato comprensibile tramite l'uso di hashToHexString.

Questa funzione tronca il byte in cui è stato tradotto ogni carattere hash in uno esadecimale; Ogni elemento è 1 byte quindi 8 bit, proprio per questo la destinazione della conversione è di lunghezza definita da

```
#define CONVERSION_SHA256_MAX (EVP_MAX_MD_SIZE *2 + 1)
```

quindi ad ogni elemento della bytestring ne corrisponde 2 esadecimali (+ 1 per il carattere di fine stringa).

```
static void hashToHexString(const unsigned char *hash, int length, char *output) {
    const char *hexChars = "0123456789abcdef"; //caratteri esadecimali
    for (int i = 0; i < length; i++) {
        /*spostamento di 4 bit del byte corrente
         * si ottiene gli ultimi 4 bit tramite l'and logico con 0xF
         * ottenendo così un valore esadecimale
         * stessa cosa si fa col successivo elemento isolando gli altri 4 bit del byte corrispondente
         *
         * L'effetto che si ha è un vero e proprio troncamento del byte in 2 valori esadecimali da 4
         * infatti 1 byte = 8 bit
         * come maschera usiamo 0xF = 1111
         */
        output[i * 2] = hexChars[(hash[i] >> 4) & 0xF];
        output[i * 2 + 1] = hexChars[hash[i] & 0xF];
    }
    output[length * 2] = '\0'; // determinare la fine della stringa
}
```

## Altre funzioni in server\_utils

### clean\_stdin

```
void clean_stdin() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}
```

Ripulisce lo Standard Input con `getchar()` leggendo un char per volta.

### utils\_strIncludeOnly

```
int utils_strIncludeOnly(char *str, char *restriction)
{
    char digit[2] = "\0";
    for(char *ptr = str; *ptr; ptr++)
    {
        digit[0] = ptr[0];
        if(strstr(restriction, digit) == NULL)
            return 0;
    }
    return 1;
}
```

Controlla se una stringa `str` è composta solo da caratteri della seconda stringa `restriction`. Scorre un puntatore attraverso `str` e vede se il primo carattere è una sottocatena di `restriction`. Se non lo è vuol dire che il carattere non è presente in `restriction`.

## utils\_isValidEmail

```
int utils_isValidEmail(char * email)
{
    char *emailDigits = "abcdefghijklmnopqrstuvwxyz_+-";
    int emlen = strlen(email);

    if(email[0] == '@' || email[0] == '.' || email[emlen-1] == '@' || email[emlen-1] == '.') return 0;

    // stringa dopo la chiocciola
    char *afterAt = strstr(email, "@");

    if(afterAt == NULL || strstr(afterAt, ".") == NULL || afterAt[1] == '.') return 0;
    afterAt++;
    if(!utils_strIncludeOnly(afterAt, emailDigits)) return 0;

    // stringa prima della chiocciola
    char *beforeAt = malloc((emlen-strlen(afterAt)-1), sizeof(char));
    memcpy(beforeAt, email, (emlen-strlen(afterAt)-1));

    if(!utils_strIncludeOnly(beforeAt, emailDigits)) return 0;

    return 1;
}
```

Prende come argomento una stringa `email` e controlla se ha le caratteristiche di un'email valida.

Come prima cosa controlla che i caratteri “@” e “.” non siano al primo posto o all'ultimo nell'email.

Cerca vede se “@” è sottostringa dell'email e salva il puntatore alla chiocciola in `afterAt`. Controlla che questo non sia nullo, che ci sia almeno un punto in `afterAt` e che subito dopo la chiocciola non ci sia un punto.

Sposta il puntatore al carattere dopo la chiocciola e controlla che `afterAt` abbia solo caratteri validi (`emailDigits`) con `utils_strIncludeOnly()`.

Identifica la stringa prima della chiocciola `beforeAt`, allocando una quantità di memoria pari alla lunghezza dell'email - la parte dopo la chiocciola - la chiocciola e inserendo i relativi caratteri con `memcpy()`.

Passa quindi `beforeAt` a `utils_strIncludeOnly()` per controllare che i caratteri siano tutti validi.

## utils\_sortByKey

È stata effettuata una piccola modifica alla funzione `sort_object()` della libreria cJSON\_Utils. È stata cambiata la stringa su cui veniva fatto il sort.

Nella libreria il sort veniva fatto su tutta la stringa generata dal `cJSON_Print()` dell'oggetto cJSON, nella nostra funzione è possibile specificare una chiave e fare il sort solo sui valori di quella chiave.

# Signal Handler

Sul server, il processo padre e i processi figlio condividono la tabella dei segnali facilitando molto la gestione dei segnali.

## SIGINT

Il segnale di interruzione è generato quando si immette la combinazione di tasti CTRL+C. La gestione delle interruzioni è delegata al metodo `customSigHandler`, presente sia nel server che nei client. L'handler si occupa di chiudere il socket prima di terminare il processo.

Server:

```
static void customSigHandler(){
    close(serverSocket);
    if(ppidServerInit == getpid()){
        printf("\nProcesso padre(%d) terminato\n",ppidServerInit);
        exit(0);
    }

    printf("\nprocesso figlio con pid %d terminato\n",getpid());
    exit(1);
}
```

Client:

```
static void customSigHandler() {
    printf("\nInterruzione ricevuta: chiusura socket.\n");
    close(clientSocket);
    exit(0);
}
```

## SIGPIPE

Il segnale `SIGPIPE` viene generato quando un processo tenta di scrivere su una connessione senza lettore.

Utilizziamo un puntatore globale al semaforo di riferimento per sbloccare tutte le sezioni critiche dei client, questo viene fatto perché nel caso di disconnessione di un client il server deve sbloccarsi.

Se un processo si disconnette in sezione critica si crea uno stallo se non si rilascia il semaforo.

Server:

```
static void customSigPipeHandler(int signo){
    printf("\nClient con pid %d disconnesso.\n",getpid());

    // se il processo era il sezione critica
    if(inCriticalSection == getpid()) {
        sem_post(*semPtr);
    }
    exit(0);
}
```

In maniera parallela nel client si effettua la disconnessione se il server non è più in ascolto.

```
static void sigpipe_handler(int signo) {
    printf("Connessione chiusa dal server.\n");
    close(clientSocket);
    exit(0);
}
```

## Gestione comportamenti anomali

### Server

#### Arresto improvviso

Se il server si arresta all'improvviso la connessione si chiude e viene segnalato al client tramite il segnale di SIGPIPE.

Il client si interrompe e viene restituito un messaggio che indica la chiusura della connessione da parte del server.

```
Server >>
-----
|           Rubrica Telefonica          |
-----
v - visita
s - ricerca
l - login admin
x - esci

Cosa vuoi fare?
>>      v

Server >>
>>
Connessione chiusa dal server.
```

L'arresto improvviso del server non comporta la perdita di dati ma è possibile che l'ultima modifica non venga salvata se l'arresto avviene prima del salvataggio sul file database.json.

### Client

#### Inserimento troppi caratteri nell'input da tastiera

L'utente può inserire solo fino a 35 caratteri in input, tutto quello che viene inserito successivamente viene eliminato e non inviato al server.

Lato client:

```
Server >>
-----
|           Rubrica Telefonica      |
-----
v - visita
s - ricerca
l - login admin
x - esci

Cosa vuoi fare?
>>     questa è una stringa davvero lunga per testare il server

Server >>
Comando non riconosciuto.
-----
|           Rubrica Telefonica      |
-----
```

Lato server:

```
Impostazioni server caricate con successo!
Server online CTRL+C per terminare
Client connesso! @ 127.0.0.1 : 4 PID: 12136 PPID: 12129
Client @ 127.0.0.1 : 4 - isAdmin: 0, Message h
Client @ 127.0.0.1 : 4 - isAdmin: 0, Message questa è una stringa davvero lung
```

### Inserimento linee di troppo durante l'attesa al semaforo

Può capitare che durante l'attesa l'utente provi ad inviare numerose righe di testo digitando caratteri e premendo invio quando il server non è pronto per ricevere messaggi.  
Queste righe vengono cancellate prima di prendere un nuovo messaggio da inviare al server.

Lato client:

```
Server >>
Qualcun altro sta modificando i contatti. Vuoi attendere? [y/N]
>>     y

prova
1
2
3
4
5

Server >>
-----
|           Contatto      |
-----
Inserire nome del contatto :    >>
```

Lato server (ciò che l'utente ha digitato non arriva al server):

```
Client @ 127.0.0.1 : 4 - User: admin
Client @ 127.0.0.1 : 4 - Password: 1234
Client @ 127.0.0.1 : 4 - Login effettuato.
Client @ 127.0.0.1 : 4 - isAdmin: 1, Message a
Client @ 127.0.0.1 : 4 in attesa per l'aggiunta di contatti.
Client @ 127.0.0.1 : 4 sta aggiungendo contatti.
```

Arresto del client in sezione critica

Attraverso il segnale di `SIGPIPE` al server viene segnalato che il client si è disconnesso. Se il pid del processo che si disconnette è uguale al processo che era in sezione critica il semaforo viene sbloccato.

Lato server:

```
Client @ 127.0.0.1 : 4 - User: admin
Client @ 127.0.0.1 : 4 - Password: 1234
Client @ 127.0.0.1 : 4 - Login effettuato.
Client @ 127.0.0.1 : 4 - isAdmin: 1, Message a
Client @ 127.0.0.1 : 4 in attesa per l'aggiunta di contatti.

Client con pid 12481 disconnesso.
Client @ 127.0.0.1 : 4 sta aggiungendo contatti.
```