Elenco Telefonico — Client/Server TCP

Studente: Dario Sella

Data: Settembre 2025

https://github.com/dariosella/elenco-telefonico

Introduzione

Per l'A.A. 2024/2025 ho sviluppato il progetto per l'esame di Sistemi Operativi tenuto dal Prof. Francesco Quaglia.

Il progetto è un applicazione client–server in C per un elenco telefonico con autenticazione, permessi (lettura/scrittura) e sincronizzazione **readers–writers** lato server.
Comunicazione via TCP, gestione dei timeout, segnali, thread e semafori POSIX.

Specifica del progetto

Realizzazione di un servizio "elenco telefonico" supportato da un server che gestisce sequenzialmente o in concorrenza (a scelta) le richieste dei client (residenti in generale su macchine diverse).

L'applicazione client deve fornire le seguenti funzioni:

1. Aggiunta di un record all'elenco telefonico (operazione accessibile solo ad utenti autorizzati).
2. Ricerca di un record all'interno dell'elenco telefonico (anche in questo caso l'operazione deve essere accessibile solo ad utenti autorizzati).

Scelte progettuali, tecniche e metodologie

Struttura del progetto

- Makefile
- `helper.h/helper.c` : wrapper I/O robusti, costanti, utility (readLine, safeSend/Recv, safeWrite, safeWait)
- `user.h/user.c` : login/registrazione e controllo permessi (file "utenti" e "permessi")
- `contact.h/contact.c` : parsing/creazione contatto, aggiungi/cerca contatto (file "rubrica")
- `server.c` : server multithread, segnale/i, RW-semafori, protocollo
- `client.c` : client interattivo con menù iniziale e menù operazioni

Costanti principali

- `TIMEOUT = 30` secondi (socket e input con allarme)
- `LISTENQ = 8` backlog listen
- Taglie buffer:
  - `BUF_SIZE = 97` (righe di input)
  - `NAME_SIZE = 64`, `NUMBER_SIZE = 32` (contatto)
  - `USR_SIZE = 32`, `PWD_SIZE = 32`, `PERM_SIZE = 16` (utente)
  - `CHOICE_SIZE = 16` (scelta dell'utente)

File dati (creati se mancanti)

Il server utilizza 3 file di testo:

- `utenti` — l'elenco di ogni utente registrato, ogni riga: **"Username Password\n"**
- `permessi` — l'elenco dei permessi per ogni utente registrato, ogni riga: **"Username permesso\n"** con **permesso** $\in \{r, w, rw\}$
- `rubrica` — l'elenco telefonico, ogni riga: **"Nome [Nomi secondari] Cognome Numero\n"**

Permessi file: `0600`.

Concorrenza

L'opzione `SO_REUSEADDR` su listening socket permette di riutilizzare subito la stessa porta dopo la chiusura del server, evitando l'errore "Address already in use".

Il server crea un thread **per client** (`pthread_create` + `pthread_detach`).

Vengono utilizzati i **cleanup handlers** in tutti i punti critici del thread (mutex, semafori, socket, heap).

Il mutex `u_mutex` sincronizza **solo** l'accesso ai file `utenti/permessi`.

I semafori `rwsem_t rw` fanno si che le letture della rubrica sono concorrenti tra loro (sezione reader), mentre gli scrittori sono esclusivi (sezione writer).

Gestione I/O

Wrapper in `helper.c`:

- `ssize_t safeRecv(int sfd, void *buffer, size_t size, int flags)`
  - accumula fino a `size`
  - `-1` → errore
  - `-2` → timeout (`EAGAIN|EWOULDBLOCK`)
  - `-3` → chiusura connessione
- `ssize_t safeSend(int sfd, const void *buffer, size_t size, int flags)`
  - usa `MSG_NOSIGNAL` per non generare `SIGPIPE`
  - `-1` → errore
  - `-2` → timeout
  - `-3` → peer chiuso (`EPIPE`)
- `ssize_t readLine(int fd, char *line, size_t size)`
  - legge fino a `\n/EOF`
  - `-1` → errore
  - `-2` → overflow della linea
- `ssize_t safeWrite(int fd, const void *buffer, size_t size)`
  - robusto a `EINTR`, scrive esattamente `size` bytes o errore

Le funzioni `handleSendReturn/handleRecvReturn` (client e server) **centralizzano la gestione degli esiti**: in caso di `-1/-2/-3` stampano, chiudono dove serve e **terminano** il thread/processo.

Gestione segnali

Server

- **Main thread**: installa handler con `sigaction`.
  - `SIGINT`, `SIGTERM` → `interruptHandler`: chiude `l_sock` e termina il processo.
  - **Ignora** `SIGPIPE`, `SIGHUP`.
- Timeout socket con `SO_RCVTIMEO/SO_SNDTIMEO` (30s).

Client

- `SIGALRM` → per input con timeout (funzione `safeInputAlarm`).
- `SIGINT`, `SIGHUP`, `SIGTERM`, `SIGQUIT` → `interruptHandler`: chiusura `c_sock` e `_exit`.
- **Ignora** `SIGPIPE`.
- Timeout socket con `SO_RCVTIMEO/SO_SNDTIMEO` (30s).

Protocollo applicativo

Per la comunicazione client-server viene utilizzato il socket del dominio `AF_INET` (IPv4) di tipo `SOCK_STREAM` con protocollo `TCP` orientato alla connessione.

Il server si mette in ascolto sulla porta indicata.

Il client si connette al server tramite l'indirizzo IP e la porta indicati; se la connessione fallisce, tenta la risoluzione DNS nel caso sia stato fornito un nome host.

Durante la comunicazione client-server, la corretta formattazione dei dati tra architetture Big Endian e Little Endian viene gestita correttamente.

1. Fase iniziale: Registrazione/Login

Il client mostra:

1. Registrarti

```
2. Loggarti
```

Invia la scelta al server come `uint32_t` in **network byte order** (`htonl`):

1. **Registrazione**:
    a. il client invia `USR_SIZE` bytes (username), `PWD_SIZE` (password), `PERM_SIZE` (permesso).
    b. Il server sincronizza l'accesso a `utenti/permessi` con `pthread_mutex_t u_mutex`, chiama `usrRegister` e risponde con **int32 (htonl)**:
        - `0` → ok
        - `-1` → errore
        - `-2` → utente già esistente
2. **Login**:
    a. il client invia username (`USR_SIZE`) e password (`PWD_SIZE`).
    b. Il server chiama `usrLogin` e risponde con **int32 (htonl)**:
        - `0` → ok
        - `-1` → errore
        - `-2` → password errata
        - `-3` → utente inesistente

Se la risposta è `0`, si passa al menù operativo.

2. Menù operativo (post-login)

Il client mostra:

```
1. Aggiungere contatto
2. Cercare contatto
3. Uscire
```

Invia la scelta al server come `uint32_t` in **network byte order** (`htonl`):

1. **Aggiungere contatto:**
    a. Server verifica `checkPermission(username, "w")` e invia **uint8_t**: `1` autorizzato, `0` negato.

Se autorizzato, il client invia una riga (`BUF_SIZE`) del tipo: **“Nome [Nomi secondari] Cognome Numero”**

b. Il server effettua la **sezione critica writer** sulla rubrica con RW-semafori (vedi sotto), esegue `addContact` e risponde con una stringa (`BUF_SIZE`) con l’esito (ok/errore/già esiste).

2. **Cercare contatto:**

a. Server verifica `checkPermission(username, "r")` e invia **uint8_t**: `1` autorizzato, `0` negato. Se autorizzato, il client invia il **nome completo del contatto** (`NAME_SIZE`).

b. Il server entra come **lettore** sulla rubrica (RW-semafori) ed esegue `searchContact`, inviando una stringa (`BUF_SIZE`) con il risultato (contatto/non trovato/errore).

3. **Uscire:** Il client chiude il socket e termina.

Le uscite dal thread usano cleanup handler (pthread_cleanup_push/pop) per:

- sbloccare `u_mutex` e RW-semafori in caso di terminazione/cancel del thread
- fare sempre `close(c_sock)` e `free(c_user)`

Sincronizzazione readers e writers sul file rubrica (Server)

Implementazione **readers–writers** con fariness per gli scrittori:

```
typedef struct {
    sem_t turnstile; // 1 nessun writer, 0 writer in attesa
    sem_t roomEmpty; // 1 stanza libera, 0 stanza occupata
    sem_t mutex;     // protegge il contatore readers
    int readers;
} rwsem_t;
```

La fairness per gli scrittori garantisce che, una volta che uno scrittore è in attesa, i lettori non lo blocchino indefinitamente, evitando la starvation e assicurando un accesso equo al file rubrica.

- **Writer** (aggiungi contatto):

1. `sem_wait(turnstile)` → blocca nuovi lettori futuri
2. `sem_wait(roomEmpty)` → attende stanza vuota
3. scrive su `rubrica`
4. `sem_post(roomEmpty)` ; `sem_post(turnstile)`

- **Reader** (cerca contatto):
  1. `sem_wait(turnstile)` ; `sem_post(turnstile)` → passa se non ci sono writer in attesa
  2. `sem_wait(mutex)` ; `readers++` ; se è il primo occupa la stanza: `sem_wait(roomEmpty)` ; `sem_post(mutex)`
  3. legge da `rubrica`
  4. `sem_wait(mutex)` ; `readers--` ; se è l'ultimo libera la stanza: `sem_post(roomEmpty)` ; `sem_post(mutex)`

Le attese sono effettuate tramite `safeWait(sem_t*)` : wrapper su `sem_wait` che ripete su `EINTR` e restituisce `-1` su errore.

Manuale d'uso

Richiede un ambiente POSIX (Linux/macOS) e GCC con pthreads e semafori POSIX.

```
make all     # compila server e client
make clean   # pulizia
```

Esecuzione

Server

```
./server -p <porta>
# es.: ./server -p 12345
```

Client

```
./client -a <indirizzo_server> -p <porta>
```

```
# es.: ./client -a 127.0.0.1 -p 12345
```
