

# Progetto Laboratorio Sistemi Operativi AA 2024/25

Autore: Tommaso Martini  
Matricola: 7048027  
Email: [tommaso.martini4@edu.unifi.it](mailto:tommaso.martini4@edu.unifi.it)  
Data di consegna: 15 luglio 2025

## Indice

<b>1</b>	<b>Progettazione</b>	<b>1</b>
1.1	Descrizione delle funzionalità ad alto livello . . . . .	1
1.2	Scelte progettuali . . . . .	2
<b>2</b>	<b>Implementazione</b>	<b>3</b>
2.1	Comunicazione client-server . . . . .	3
2.2	Autenticazione e privilegi . . . . .	3
2.3	Gestione ticket . . . . .	4
<b>3</b>	<b>Compilazione ed Esecuzione</b>	<b>4</b>
3.1	Makefile e compilazione . . . . .	4
3.2	Esecuzione . . . . .	5
3.3	Esempi di esecuzione . . . . .	5

## 1 Progettazione

La struttura del progetto è organizzata secondo principi di modularità e separazione delle responsabilità, al fine di ottenere un'architettura chiara, manutenibile ed estensibile. In particolare, è stata adottata una convenzione che distingue tra:

- *intestazioni pubbliche*, contenute nella directory `include/`, ovvero file header destinati a essere esposti e utilizzati da moduli esterni,
- *sorgenti e intestazioni private*, collocati nella directory `src/`, che racchiudono i dettagli implementativi interni ai vari moduli.

Questa soluzione permette di incapsulare le strutture dati e le funzionalità interne, mantenendo un controllo esplicito sull'interfaccia pubblica del progetto, migliorando la coesione interna dei moduli e facilitando eventuali estensioni future.

L'architettura del sistema è suddivisa in due tipologie principali di moduli:

- *moduli di comunicazione*: responsabili della gestione delle interazioni tra le varie entità del sistema (cartelle `src/net`, `src/client` e `src/server`)
- *moduli di libreria*: che incapsulano logiche funzionali riutilizzabili all'interno del progetto (cartelle `src/auth` e `src/ticket`)

### 1.1 Descrizione delle funzionalità ad alto livello

I moduli di gestione della comunicazione si occupano di tutte le operazioni di lettura e scrittura su socket, nonché della creazione e gestione delle strutture dati utilizzate per rappresentare lo stato del sistema.

Per strutturare in modo ordinato il flusso di comunicazione, è stato implementato un meccanismo di smistamento delle richieste, sia lato server che lato client, tramite i moduli `server-dispatcher` e

`client-dispatcher`. Questi moduli si occupano di inoltrare opportunamente i messaggi in ingresso verso le funzioni di gestione corrispondenti.

La comunicazione tra client e server avviene tramite lo scambio di *messaggi*, contenuti nelle due strutture simmetriche richiesta-risposta. In particolare:

- L'interfaccia client richiede all'utente l'inserimento di un *codice di operazione*. Tale codice viene interpretato da appositi gestori di input, che determinano il tipo di richiesta da inviare al server.
- Una volta raccolti eventuali parametri tramite prompt, viene costruito l'oggetto richiesta e serializzato sulla socket.
- Il server riceve il messaggio, lo deserializza e ricostruisce la richiesta, identificandone il tipo attraverso il codice operazione. La richiesta viene quindi inoltrata ai gestori interni, che la smistano verso le funzioni di libreria appropriate.
- Dopo l'elaborazione, viene generata una risposta contenente i risultati dell'operazione. Tale risposta viene inviata al client, che la interpreta e presenta le informazioni all'utente tramite output a schermo.

Il sistema offre funzionalità quali:

- aggiunta, rimozione e ricerca (anche filtrata) di ticket nella lista
- operazioni di autenticazione, come login e logout degli utenti
- salvataggio e caricamento dello stato completo dei ticket/utenti su file, garantendo la persistenza dei dati
- gestione delle interruzioni: il sistema intercetta i segnali di interruzione (es. SIGINT) e, in risposta, esegue un gestore dell'interruzione prima di terminare l'esecuzione
- gestione della terminazione ordinata, che permette al client di inviare un codice di uscita per chiudere correttamente la comunicazione.

Questo ciclo di comunicazione e gestione delle richieste continua fino a quando il client non richiede esplicitamente la chiusura o fino a che la comunicazione viene interrotta.

## 1.2 Scelte progettuali

Nel progetto è stata adottata una politica di gestione della memoria *centralizzata*, limitata a specifiche funzioni di libreria responsabili sia dell'allocazione che della deallocazione delle strutture dati. Ho ritenuto questa scelta coerente con i principi di *incapsulamento* e *separazione delle responsabilità*: poiché le strutture dati interne non vengono esposte ai moduli esterni, la loro gestione completa — inclusi creazione, utilizzo e distruzione — è responsabilità esclusiva della libreria.

In particolare, ogni oggetto dinamico viene creato tramite apposite funzioni fornite dalla libreria e viene liberato esclusivamente da essa. Questo approccio garantisce un controllo rigoroso sull'ownership della memoria, riducendo il rischio di memory leak o accessi non validi da parte del codice client.

Per rappresentare lo stato del sistema e manipolare dati complessi, è stato fatto ampio uso delle **struct** di C. Queste strutture permettono di modellare in modo chiaro e compatto oggetti complessi come utenti, messaggi, richieste e risposte, ticket e qualsiasi altra entità del dominio applicativo. Le **struct** facilitano inoltre la serializzazione e deserializzazione dei dati, operazioni fondamentali nella comunicazione client-server.

Infine per quanto riguarda le strutture dati dinamiche, si è fatto affidamento principalmente su *liste concatenate*, che offrono flessibilità nella gestione dinamica degli elementi e consentono operazioni di inserimento e rimozione efficienti, particolarmente utili per implementare meccanismi di gestione dei ticket.

## 2 Implementazione

### 2.1 Comunicazione client-server

Il client inoltra le richieste tramite un *dispatcher* definito come segue:

```
typedef struct dispatcher {
    int code;
    input_handler fn;
} dispatcher_t;
```

Un *input handler* è una funzione con la firma:

```
typedef message_t *(*input_handler)(FILE *f);
```

che viene mappata in una tabella di dispatch gestita dal modulo client.

L'input handler si occupa di raccogliere i dati di input e costruire un messaggio, rappresentato dalla struttura:

```
typedef struct message {
    uint32_t size;
    char **content;
} message_t;
```

Questo consente al client di comporre la richiesta da inviare al server, definita come:

```
typedef struct request {
    RequestCode code;
    message_t *payload;
} request_t;
```

Il server, una volta ricevuta la richiesta, la deserializza e la inoltra ai gestori specifici:

```
typedef struct dispatcher {
    int code;
    handler fn;
} dispatcher_t;
```

dove *handler* è una funzione con la firma:

```
typedef response_t *(*handler)(session_t *, message_t *);
```

I gestori elaborano la richiesta invocando le funzioni di libreria e producono una risposta definita da:

```
typedef struct response {
    ResponseCode code;
    message_t *payload;
} response_t;
```

### 2.2 Autenticazione e privilegi

Lo stato di un utente è rappresentato dalla struttura:

```
typedef struct user {
    uint32_t uid;
    char *username;
    char *password;
    Privileges privileges;
    struct user *next;
} user_t;
```

Per tenere traccia dei privilegi, è stata scelta una **bitmask**, in quanto semplice da implementare e efficiente nelle operazioni bitwise. I privilegi sono definiti tramite enum e assegnati con macro di supporto:

```
#define BIT(n) (1 << (n))

typedef enum Privileges {
    PRIVILEGES_ADMIN      = BIT(0),
    PRIVILEGES_GUEST      = BIT(1),
    PRIVILEGES_SUPPORT_AGENT = BIT(2)
} Privileges;
```

Lo stato della sessione e dell'autenticazione è mantenuto nella struttura:

```
typedef struct session {
    int fd;
    struct user *user;
    int logged_in;
    Privileges privileges;
} session_t;
```

dove fd rappresenta l'handle della socket.

## 2.3 Gestione ticket

Lo stato di un ticket è definito dalla seguente struttura:

```
typedef struct ticket {
    uint32_t tid;
    char *title;
    char *description;
    char *date;           // formato YYYY-MM-DD
    TicketPriority priority;
    TicketStatus status;
    struct user *support_agent;
    struct ticket *next;
} ticket_t;
```

Tra le operazioni sulla lista dei ticket, una delle più interessanti è la ricerca filtrata, implementata tramite filtri personalizzabili.

Un filtro è definito come:

```
typedef int (*ticket_filter)(const ticket_t *target, va_list args);
```

mentre la funzione di ricerca principale ha la firma:

```
int _get_tickets(ticket_t ***destination, ticket_filter filter, ...);
```

Questo approccio permette di aggiungere nuovi filtri semplicemente definendo nuove funzioni filtro, senza dover modificare la funzione principale di ricerca, garantendo così un'architettura modulare e facilmente estensibile.

## 3 Compilazione ed Esecuzione

La compilazione dei sorgenti del progetto è automatizzata tramite un **Makefile** locato nella root della cartella del progetto. Gestisce in particolare la compilazione separata dei tre componenti principali:

- **server**: il server principale
- **client**: l'interfaccia client
- **client handler**: un modulo interno che rappresenta un'istanza del server e gestisce la connessione client-server

### 3.1 Makefile e compilazione

CC e CFLAGS definiscono rispettivamente il compilatore (gcc) e i percorsi delle intestazioni, tramite flag `-Ipercorso/`, che istruiscono il compilatore su dove cercare i file intestazione.

Vengono poi definiti i **target di compilazione** per ciascun modulo del progetto, individuando automaticamente i sorgenti .c mediante l'uso delle funzioni wildcard e filter-out.

Ad esempio:

```
$(SERVER_TARGET): $(SERVER_OBJ)
    $(CC) $(CFLAGS) -o $@ $^
```

Questa regola compila il target indicato da SERVER.TARGET (cioè *server*) utilizzando tutti i file oggetto elencati in SERVER.OBJ. In particolare:

- \$@ rappresenta il nome del target, cioè il file da generare;
- \$^ rappresenta tutti i prerequisiti, cioè i file oggetto necessari alla compilazione.

In questo modo la compilazione è automatizzata e mantiene separati i moduli del progetto in modo ordinato, estensibile e facilmente manutenibile.

Vengono definite quattro regole principali: **all**, **install**, **clean**, **setup**

**make setup** è un'istruzione che ho pensato di fornire per rendere la compilazione più immediata e pulita. Oltre a eseguire la sequenza predefinita **make clean -> make all -> make install**, si occupa di registrare l'intero processo in un file di log. In caso di errori, stamperà a schermo le ultime righe del log per semplificare il debugging. Mostra infine un messaggio con le istruzioni per eseguire i moduli server e client.

L'output prodotto a schermo sarà del tipo:

```
user@ubuntu:~$ make setup
[SETUP] Cleaning previous build...
[SETUP] Compiling all targets...
[SETUP] Installing binaries and directories...
[SETUP] Loading test users...
[SETUP] Done. Full log in log/setup.log
You can now run with ./bin/server <port> and ./bin/client <ip> <port>
```

## 3.2 Esecuzione

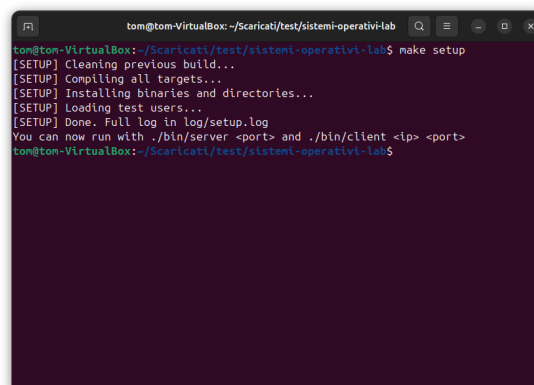
La compilazione produrrà alcuni eseguibili tra cui server e client. Per avviare ciascun modulo è sufficiente digitare (assumendo di trovarsi nella root della cartella del progetto):

**./bin/server** e in un altro terminale **./bin/client**

**Nota:** il modulo **client\_handle** non dovrebbe essere eseguito direttamente. È pensato come modulo interno chiamato automaticamente allo stabilirsi della connessione.

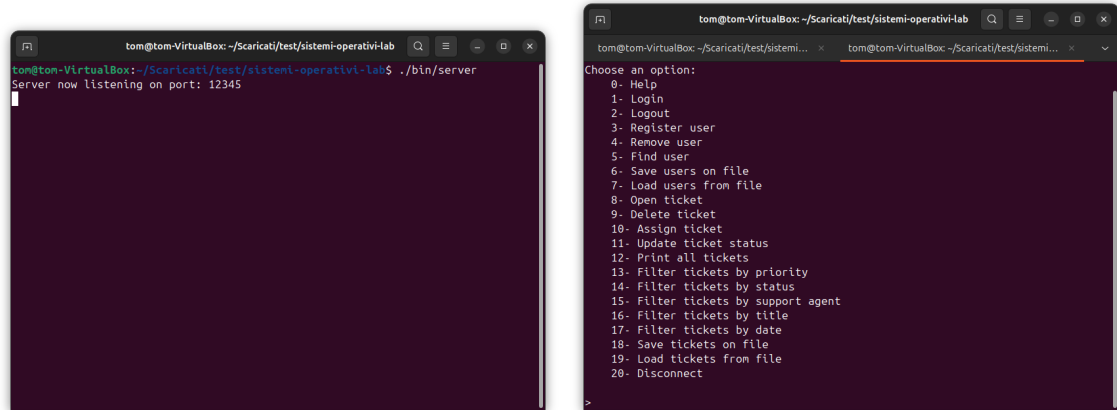
## 3.3 Esempi di esecuzione

Inizialmente si compila con **make setup**



```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
tom@tom-VirtualBox:~/Scaricati/test/sistemi-operativi-lab$ make setup
[SETUP] Cleaning previous build...
[SETUP] Compiling all targets...
[SETUP] Installing binaries and directories...
[SETUP] Loading test users...
[SETUP] Done. Full log in log/setup.log
You can now run with ./bin/server <port> and ./bin/client <ip> <port>
tom@tom-VirtualBox:~/Scaricati/test/sistemi-operativi-lab$
```

Eseguendo il server va in ascolto e il client stampa un menu di scelta

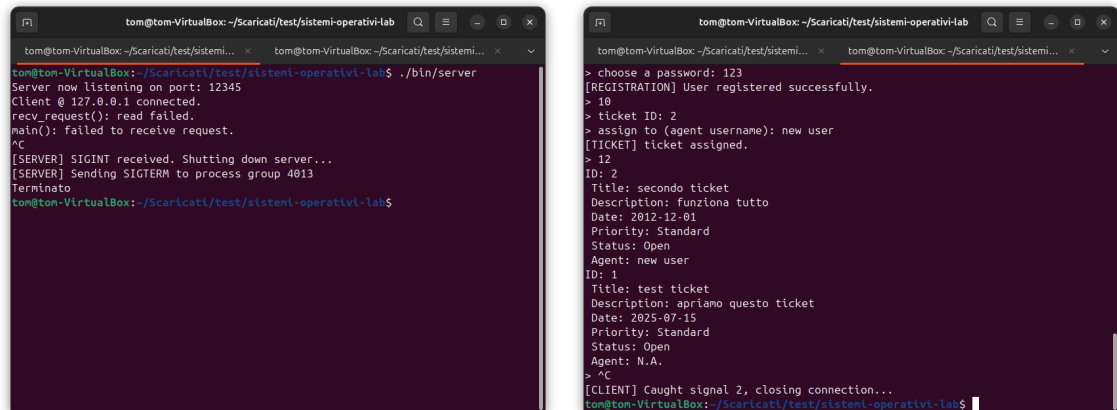


The image shows two terminal windows. The left window is the server, and the right window is the client.

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
tom@tom-VirtualBox:~/Scaricati/test/sistemi-operativi-lab$ ./bin/server
Server now listening on port: 12345
```

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
Choose an option:
0- Help
1- Login
2- Logout
3- Register user
4- Remove user
5- Find user
6- Save users on file
7- Load users from file
8- Open ticket
9- Delete ticket
10- Assign ticket
11- Update ticket status
12- Print all tickets
13- Filter tickets by priority
14- Filter tickets by status
15- Filter tickets by support agent
16- Filter tickets by title
17- Filter tickets by date
18- Save tickets on file
19- Load tickets from file
20- Disconnect
>
```

Se arriva SIGINT, il server esce in modo controllato e fa terminare tutti i figli (precedentemente inseriti nello stesso gruppo per tenerne traccia) così come quando è il client a riceverlo:

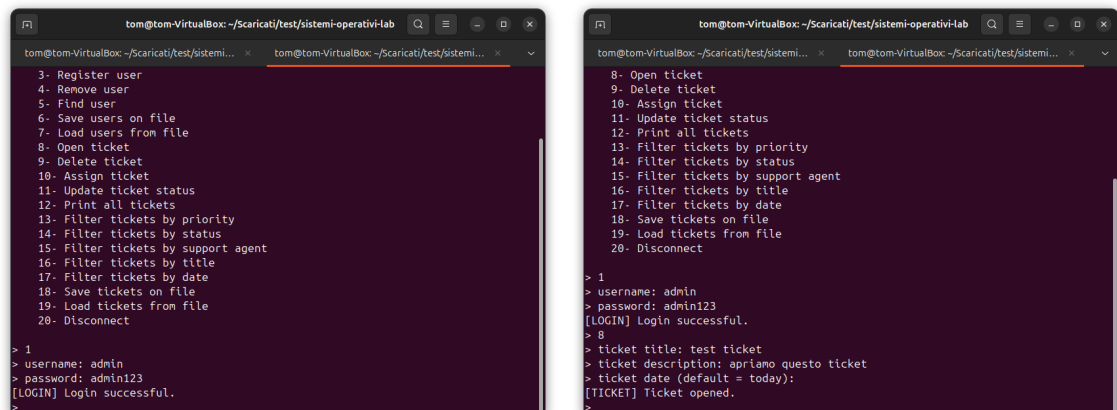


The image shows two terminal windows. The left window is the server, and the right window is the client.

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
tom@tom-VirtualBox:~/Scaricati/test/sistemi-operativi-lab$ ./bin/server
Server now listening on port: 12345
Client @ 127.0.0.1 connected.
recv_request(): read failed.
main(): failed to receive request.
^C
[SERVER] SIGINT received. Shutting down server...
[SERVER] Sending SIGTERM to process group 4013
Terminato
tom@tom-VirtualBox:~/Scaricati/test/sistemi-operativi-lab$
```

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
> choose a password: 123
[REGISTRATION] User registered successfully.
> 10
> ticket ID: 2
> assign to (agent username): new user
[TICKET] ticket assigned.
> 12
ID: 2
Title: secondo ticket
Description: funziona tutto
Date: 2012-12-01
Priority: Standard
Status: Open
Agent: new user
ID: 1
Title: test ticket
Description: apriamo questo ticket
Date: 2025-07-15
Priority: Standard
Status: Open
Agent: N.A.
> ^C
[CLIENT] Caught signal 2, closing connection...
tom@tom-VirtualBox:~/Scaricati/test/sistemi-operativi-lab$
```

Effettuo il login e inserisco un ticket

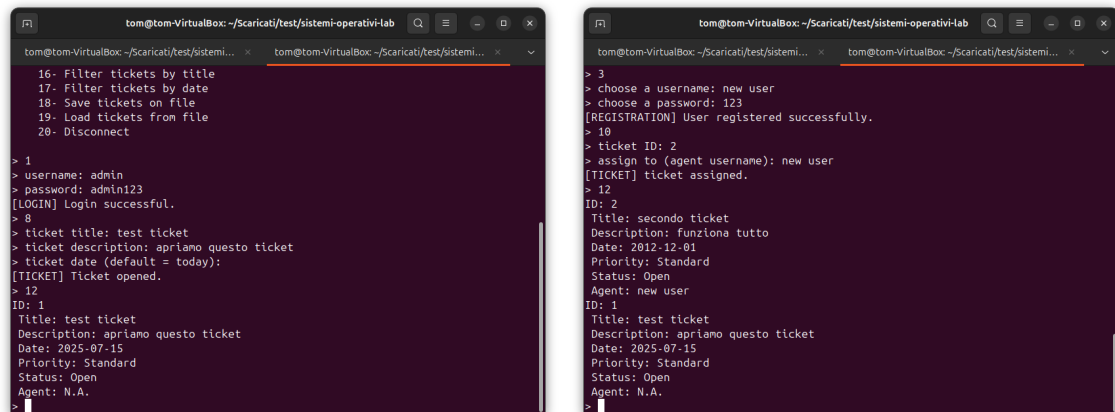


The image shows two terminal windows. The left window is the server, and the right window is the client.

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
3- Register user
4- Remove user
5- Find user
6- Save users on file
7- Load users from file
8- Open ticket
9- Delete ticket
10- Assign ticket
11- Update ticket status
12- Print all tickets
13- Filter tickets by priority
14- Filter tickets by status
15- Filter tickets by support agent
16- Filter tickets by title
17- Filter tickets by date
18- Save tickets on file
19- Load tickets from file
20- Disconnect
> 1
> username: admin
> password: admin123
[LOGIN] Login successful.
>
```

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
8- Open ticket
9- Delete ticket
10- Assign ticket
11- Update ticket status
12- Print all tickets
13- Filter tickets by priority
14- Filter tickets by status
15- Filter tickets by support agent
16- Filter tickets by title
17- Filter tickets by date
18- Save tickets on file
19- Load tickets from file
20- Disconnect
> 1
> username: admin
> password: admin123
[LOGIN] Login successful.
> 8
> ticket title: test ticket
> ticket description: apriamo questo ticket
> ticket date (default = today):
[TICKET] Ticket opened.
>
```

Stampo i ticket e ne assegno uno ad un utente



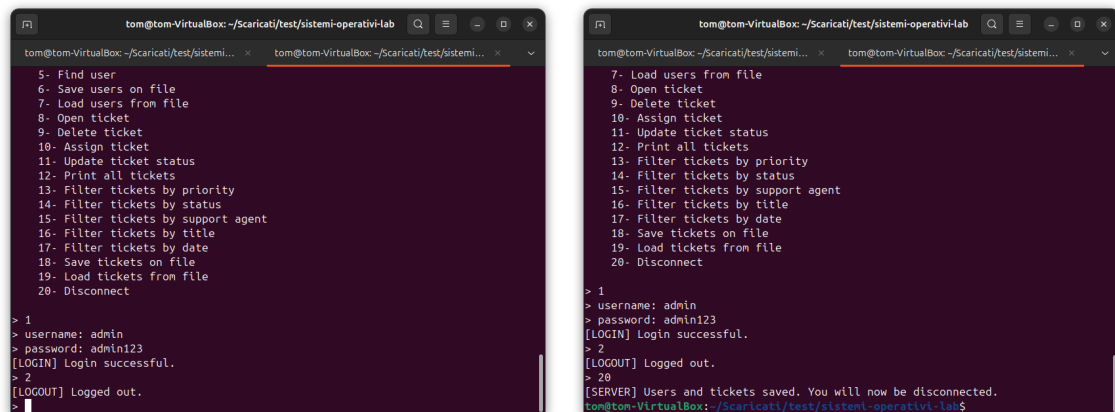
The first terminal window shows a menu with options 16-20. Option 16 is selected, leading to a login prompt. The user enters 'admin' and 'admin123', and the login is successful. Then, option 8 is selected, leading to a ticket creation prompt. The user enters 'test ticket' for the title and 'apriamo questo ticket' for the description. The ticket is opened with ID 1. The second terminal window shows option 3 selected, leading to a user registration prompt. The user enters 'new user' for the username and '123' for the password. The user is registered successfully. Then, option 10 is selected, leading to a ticket assignment prompt. The user enters 'new user' for the agent username. The ticket is assigned with ID 2. The output shows details for both tickets.

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
16- Filter tickets by title
17- Filter tickets by date
18- Save tickets on file
19- Load tickets from file
20- Disconnect

> 1
> username: admin
> password: admin123
[LOGIN] Login successful.
> 8
> ticket title: test ticket
> ticket description: apriamo questo ticket
> ticket date (default = today):
[TICKET] Ticket opened.
> 12
ID: 1
Title: test ticket
Description: apriamo questo ticket
Date: 2025-07-15
Priority: Standard
Status: Open
Agent: N.A.

tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
> 3
> choose a username: new user
> choose a password: 123
[REGISTRATION] User registered successfully.
> 10
> ticket ID: 2
> assign to (agent username): new user
[TICKET] ticket assigned.
> 12
ID: 2
Title: secondo ticket
Description: funziona tutto
Date: 2012-12-01
Priority: Standard
Status: Open
Agent: new user
ID: 1
Title: test ticket
Description: apriamo questo ticket
Date: 2025-07-15
Priority: Standard
Status: Open
Agent: N.A.
```

Effettuo logout e chiudo



The first terminal window shows a menu with options 5-20. Option 1 is selected, leading to a login prompt. The user enters 'admin' and 'admin123', and the login is successful. Then, option 2 is selected, leading to a logout prompt. The user is logged out successfully. The second terminal window shows a menu with options 7-20. Option 7 is selected, leading to a user loading prompt. The user enters 'admin' and 'admin123', and the user is loaded successfully. Then, option 20 is selected, leading to a shutdown prompt. The user is disconnected and the system is shutdown.

```
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
5- Find user
6- Save users on file
7- Load users from file
8- Open ticket
9- Delete ticket
10- Assign ticket
11- Update ticket status
12- Print all tickets
13- Filter tickets by priority
14- Filter tickets by status
15- Filter tickets by support agent
16- Filter tickets by title
17- Filter tickets by date
18- Save tickets on file
19- Load tickets from file
20- Disconnect

> 1
> username: admin
> password: admin123
[LOGIN] Login successful.
> 2
[LOGOUT] Logged out.

tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab
7- Load users from file
8- Open ticket
9- Delete ticket
10- Assign ticket
11- Update ticket status
12- Print all tickets
13- Filter tickets by priority
14- Filter tickets by status
15- Filter tickets by support agent
16- Filter tickets by title
17- Filter tickets by date
18- Save tickets on file
19- Load tickets from file
20- Disconnect

> 1
> username: admin
> password: admin123
[LOGIN] Login successful.
> 2
[LOGOUT] Logged out.
[SERVER] Users and tickets saved. You will now be disconnected.
tom@tom-VirtualBox: ~/Scaricati/test/sistemi-operativi-lab$
```