# Tesina Sistemi Operativi

FLAVIO SIMONELLI

## Progetto: Servizio di Messaggistica in UNIX

### Specifiche del Progetto

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 risiede 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 potrà essere utilizzato solo da utenti autorizzati (deve essere quindi previsto un meccanismo di autenticazione).

### Discussione delle scelte di progetto e realizzative, e delle tecniche e metodologie generali usate

Il progetto che ho sviluppato è un'applicazione di messaggistica implementata in linguaggio C, composta da un client e un server.

Il sistema ha l'obiettivo di mettere in comunicazione gli utenti registrati con i loro client tramite il server, che ha il compito di gestire i messaggi inviati e gli utenti registrati.

Il server ha la responsabilità di mantenere i dati relativi alle credenziali degli utenti e i loro rispettivi messaggi in arrivo.

#### MULTI_THREADING per il server:

Il thread principale del server assume un ruolo cruciale nell'inizializzazione e nella gestione delle strutture dati, oltre alla configurazione della socket per l'ascolto delle connessioni. Una volta completate queste operazioni, il thread principale entra in uno stato di ascolto, pronto a ricevere nuove connessioni da parte dei client.

Appena un client si connette, il server attiva un nuovo thread dedicato a gestire le interazioni con quel particolare client. Questo approccio multi-threading consente al server di gestire simultaneamente più connessioni senza compromettere le prestazioni. Ogni thread associato a un client gestisce in modo indipendente le richieste provenienti da quel client specifico, migliorando l'efficienza complessiva del sistema.

Questo modello di progettazione favorisce la scalabilità, consentendo al server di gestire concorrentemente più utenti senza interrompere le operazioni per altri client connessi. Inoltre, il fatto che il thread principale si rimetta in ascolto dopo aver creato un nuovo thread per un client garantisce che il server sia sempre pronto ad accettare nuove connessioni, garantendo una risposta immediata a eventuali richieste in arrivo.

### SINGLE_THREADING per il client:

Nel contesto del client, invece, l'approccio single-threaded è più adeguato.

Il client, una volta avviato, gestisce le sue operazioni in modo sequenziale, interagendo con l'utente e inviando/ricevendo messaggi dal server.

Il modello single-threaded semplifica notevolmente la gestione delle operazioni del client, evitando la complessità associata all'implementazione di concorrenza e thread multipli. Questa scelta di progettazione è spesso sufficiente per applicazioni di messaggistica leggere in cui il client è principalmente impegnato nell'interazione con l'utente e nell'invio/ricezione di messaggi, come in questo caso.

### CONNESSIONE attraverso socket:

La connessione fra client e server è attraverso una socket il cui indirizzo è cablato all'interno del codice.

L'indirizzo della socket cablato direttamente nel codice del client rappresenta un approccio conveniente per l'utente, poiché lo solleva da ogni responsabilità relativa alla configurazione manuale dell'indirizzo di connessione. Questa scelta di design semplifica notevolmente l'esperienza dell'utente, riducendo la possibilità di errori di configurazione.

Il client, quindi, utilizza un indirizzo di socket predefinito, eliminando la necessità per l'utente di inserire manualmente l'indirizzo di connessione ogni volta che avvia l'applicazione. Questo è particolarmente utile per utenti meno esperti o per ridurre il carico operativo in ambienti in cui l'indirizzo di connessione non cambia frequentemente.

Tuttavia, è importante notare che questa comodità potrebbe limitare la flessibilità in scenari in cui gli utenti desiderano connettersi a server con indirizzi diversi. Pertanto, la scelta di cablare l'indirizzo nel codice dovrebbe essere ponderata in base ai requisiti specifici dell'applicazione e all'esperienza dell'utente che si desidera offrire.

### SICUREZZA tramite libsodium:

La sicurezza dei messaggi trasmessi all'interno della socket è prioritaria nel mio progetto, e per garantire un livello elevato di protezione, ho implementato un algoritmo a chiave mista utilizzando la libreria libsodium.

La mia scelta deriva dal fatto che:

- libsodium è stata sviluppata da esperti di crittografia ed è basata sulla ben consolidata libreria NaCl (Networking and Cryptography library). La sua robustezza è stata ampiamente testata e verificata.
- La libreria è progettata per essere resistente ad attacchi comuni, come attacchi temporizzati (side-channel attacks) e altre vulnerabilità crittografiche.
- libsodium è distribuita con una licenza libera (ISC license), che consente un'ampia adozione in progetti open source e commerciali.

Attraverso la libreria libsodim ho applicato l'algoritmo argon2 per le password.

**Argon2** è un algoritmo di hash delle password, noto per essere resilienti contro attacchi di tipo brute-force, attacchi basati su dizionari e attacchi di tipo side-channel. Argon2 è stato progettato per essere una scelta sicura e moderna per l'hashing delle password e viene comunemente utilizzato per la memorizzazione sicura delle password in modo da proteggerle dagli attacchi informatici.

Per il trasferimento di dati fra client e server ho invece applicato un algoritmo a chiave mista.

La **crittografia a chiave mista**, o crittografia ibrida, è un approccio che combina i vantaggi della crittografia simmetrica e asimmetrica all'interno di un sistema crittografico. Questo approccio è spesso utilizzato per sfruttare le forze di entrambi i tipi di crittografia, superando le limitazioni individuali.

In questo approccio, una coppia di chiavi viene generata: una chiave pubblica e una chiave privata. La chiave pubblica cifra il messaggio, mentre la chiave privata decifra. Per garantire efficienza, viene utilizzata anche una chiave simmetrica generata casualmente per cifrare il messaggio. Questa chiave simmetrica è a sua volta cifrata con la chiave pubblica del destinatario e inviata insieme al messaggio cifrato. Il destinatario, possedendo la chiave privata corrispondente, decifra la chiave simmetrica e poi utilizza questa chiave per decifrare il messaggio.

### Strutture Dati utilizzate:

Per la gestione dei dati, ho implementato una struttura dati basata su **liste concatenate**.

La lista degli utenti è semplice, con un puntatore alla testa, mentre per i messaggi ho adottato un approccio più complesso. Utilizzo un puntatore sia all'inizio che alla fine della lista dei messaggi per ottimizzare i costi di inserimento, garantendo efficienza nell'aggiunta di nuovi messaggi. Questa scelta di design contribuisce a mantenere prestazioni ottimali nel trattare grandi volumi di messaggi.

### Archiviazione dei dati quando il server viene chiuso:

Per mantenere una coerenza dei dati anche quando il server viene chiuso ho scelto di **salvare tutti i dati all'interno dei file** così nominati:

Credential.txt = per il file relativo a utente e password

Nome_Utente.txt = per i file relativi ai messaggi in arrivo per un determinato utente

### Gestione dei segnali

Il mio server è progettato con una gestione avanzata dei segnali per garantire la robustezza e la coerenza dei dati durante il suo funzionamento.

In particolare, il server blocca tutti i segnali in entrata, assicurando un controllo completo sulle azioni da intraprendere in risposta a segnali specifici. Nel caso di un'interruzione del processo da parte dell'utente tramite **SIGINT** (ad esempio, con Ctrl+C), il server è in grado di rilevare questo segnale e attivare una procedura che permette di salvare tutti i dati critici su file e chiudere la socket in maniera ordinata, garantendo la coerenza delle informazioni prima della terminazione del processo. Inoltre, per garantire che la chiusura del terminale non comporti la chiusura inattesa del server, è stato implementato un meccanismo per gestire il segnale **SIGCLOSE**. In questo modo, il server continua a funzionare in modo stabile anche se il terminale

viene chiuso, evitando interruzioni indesiderate e fornendo una maggiore affidabilità nell'esecuzione delle sue funzionalità.

Questa attenta gestione dei segnali contribuisce a preservare l'integrità dei dati e a garantire una chiusura controllata del server, mantenendo la coerenza del sistema anche in situazioni inaspettate.

Per quanto riguarda il client è stato invece utilizzato proprio il segnale **SIGINT** per permettere la corretta chiusura del processo.

### Approccio Object-Oriented

Nel mio progetto, ho adottato un approccio object-oriented nella progettazione delle funzioni sia per il server che per il client. Un elemento distintivo di questa implementazione è l'utilizzo di valori di ritorno codificati, dove le funzioni restituiscono un codice di errore 0 in assenza di problemi, altrimenti un numero diverso da 0 per segnalare eventuali errori. Questa scelta di progettazione mira a migliorare la gestione degli errori consolidando la logica di gestione in un unico punto del codice. Grazie a questo approccio, ho notato una chiarezza notevole nel flusso di controllo, poiché posso concentrarmi sulla gestione degli errori in una sezione specifica del programma. Oltre a rendere il codice più chiaro, questo metodo offre una maggiore modularità, semplificando la manutenzione del codice.

La gestione centralizzata degli errori riduce la complessità complessiva del progetto, facilitando il debugging e consentendo modifiche più agevoli alla logica di gestione degli errori.

## Breve manuale d'uso dei programmi (come compilare, come installare)

Le scelte progettuali precedentemente indicate hanno portato ad una semplice realizzazione di questa fase.

Per compilare vi è l'obbligo di aver installato la libreria libsodium.

Questo può essere fatto da terminale utilizzando il makefile integrato per sistemi UNIX basati su Debian, attraverso le direttive:

- Make client
- Make server

Per sistemi UNIX generici invece bisogna installare autonomamente la libreria facendo riferimento alla pagina ufficiale, e successivamente compilare tramite le direttive:

- Make client_nocheck
- Make server_nocheck

Verranno creati due file eseguibili omonimi.

### Utilizzo del server

L'utilizzo del server è banale poiché una volta avviato sarà tutto automatico.

L'unica cosa che si può fare è chiuderlo attraverso il comando (Ctrl+c).

### Utilizzo del client

Appena dopo lo start del client esso ci notificherà la corretta connessione con il server e subito dopo ci darà la possibilità di selezionare una fra le seguenti opzioni:

- Creare un nuovo account
- Autenticarsi
- Rimuovere il proprio account

Una volta loggati si avranno la possibilità di scegliere se:

- Inviare un messaggio
- Leggere i messaggi in arrivo
- Eliminare un messaggio in arrivo

## Tutti i sorgenti del progetto

I file per eseguire correttamente il progetto sono:

- Makefile
- Client_unix.c
- Client_unix.h
- Server.c
- Server.h
- Inpute_utente.c
- Input.utente.h
- Transfert_socket.c
- Transfert_socket.h