# Specifica del Progetto: Sistema di Bacheca Client-Server

## Introduzione: Architettura e Scelte Progettuali Chiave

Questo documento costituisce la specifica tecnica completa e l'analisi di progettazione per un'applicazione di bacheca basata su un'architettura client-server. Sviluppato interamente in linguaggio C per sistemi operativi POSIX, il progetto si fonda su un insieme di scelte architetturali mirate a garantire robustezza e efficienza. L'analisi che segue non si limiterà a descrivere il funzionamento del sistema, ma esplorerà il "perché" dietro ogni decisione implementativa.

Il progetto "poggia" su tre pilastri logici fondamentali:

1. **Gestione della Concorrenza tramite un Pool di Thread (Thread Pool):** Per servire un numero potenzialmente elevato di client simultaneamente è stato implementato un pool di thread a dimensione fissa. Questo modello pre-alloca un numero definito di thread "worker" che prelevano le richieste da una coda di task concorrente. Questa scelta offre un controllo sulle risorse di sistema, previene l'esaurimento dei thread in condizioni di carico elevato e migliora la latenza media delle richieste eliminando l'overhead di creazione/distruzione dei thread per ogni connessione.
2. **Protocollo di Comunicazione:** La comunicazione tra client e server è governata da un protocollo binario custom. Ogni pacchetto è preceduto da un `header` a dimensione fissa che ne definisce il tipo e la lunghezza del payload. Per superare la natura di "stream" di TCP sono state implementate delle funzioni (`send_all`, `recv_all`) che assicurano la trasmissione e la ricezione atomica di ogni pacchetto a livello applicativo.
3. **Persistenza dei Dati tramite "In-Memory Cache" con Strategia "Save-on-Shutdown":** Per massimizzare le performance e minimizzare la latenza, l'intero stato della bacheca (messaggi e utenti) viene mantenuto in memoria RAM durante l'esecuzione del server. L'accesso al disco, operazione notoriamente lenta, è limitato a due momenti critici: il caricamento iniziale dei dati all'avvio del server (load) e il salvataggio completo dello stato in memoria durante la procedura di chiusura controllata (shutdown), garantendo elevata efficienza ma con il compromesso consapevole di una potenziale perdita di dati in caso di crash improvviso non gestito.

Queste scelte implementative consentono di sviluppare un'interfaccia client-server sostenibile, dal punto di vista dell'ottimizzazione delle risorse, ma soprattutto che garantisca una user-experience accettabile e conforme con la richiesta progettuale.

**Specifiche Funzionali Chiave:**

1. **Gestione Utenti**: Il sistema permette a nuovi utenti di registrarsi con un nome utente e una password. Gli utenti esistenti possono accedere al sistema fornendo le proprie credenziali.
2. **Persistenza dei Dati**: Tutte le informazioni, inclusi gli account utente e i messaggi sulla bacheca, vengono salvate su file, garantendone la conservazione tra diverse sessioni di esecuzione del server.

3. **Concorrenza**: Il server è in grado di gestire le richieste di più client contemporaneamente. L'accesso alle risorse condivise (come l'archivio dei messaggi e il file degli utenti) è gestito in modo sicuro tramite meccanismi di mutua esclusione per prevenire race condition.
4. **Interazione con la Bacheca**:
    - **Visualizzazione**: Gli utenti autenticati possono richiedere e visualizzare l'intero contenuto della bacheca. I messaggi sono organizzati e visualizzati in ordine cronologico.
    - **Pubblicazione**: Gli utenti possono pubblicare nuovi messaggi, specificando un oggetto e un corpo.
    - **Cancellazione**: Un utente può cancellare solo i messaggi di cui è l'autore.
5. **Robustezza**: Il sistema implementa meccanismi per gestire la disconnessione anomala dei client e per tentare una chiusura pulita (shutdown) del server in risposta a segnali esterni (come SIGINT).

---

# MANUALE D'USO

**Compilazione(gcc compiler):**

Per compilare il progetto basta eseguire sulla riga di comando del terminale

```
make all
```

Mentre per eseguire la pulizia degli eseguibili basta scrivere

```
make clean
```

**Esecuzione:**

**1. Avviare il Server:**

Per avviare il server, eseguire il seguente comando nel terminale:

```
./server_executable
```

Il server si metterà in ascolto sulla porta 8080 (come definito in server.c e client.c) e creerà una directory data nella sua posizione corrente. All'interno di questa directory verranno salvati i file users.txt e messages.txt per la persistenza dei dati. Il terminale mostrerà un messaggio di conferma:

**2. Avviare il Client:**

Per connettersi al server, aprire un nuovo terminale e avviare il client.

Se il server è in esecuzione sulla stessa macchina usare il comando:

```
./client_executable
```

**3. Interagire con l'applicazione:**

Una volta connesso, il client mostrerà un menu testuale per registrarsi, accedere o uscire. Dopo aver effettuato l'accesso, sarà disponibile un secondo menu per visualizzare la bacheca, postare un messaggio o cancellarne uno.

**4. Chiudere il Server:**

Per chiudere il server in modo pulito, premere `Ctrl+C` nel terminale in cui è in esecuzione. Il server intercetterà il segnale e avvierà una procedura di "shutdown": attenderà che tutti i client connessi terminino le loro operazioni prima di salvare i dati e chiudersi.

---

# GESTIONE DELLO STATO E DELLA DURABILITA' DEI DATI

La permanenza dei dati è una caratteristica fondamentale per un sistema come una bacheca, dove le informazioni (utenti e messaggi) devono sopravvivere ai riavvii del server. Questo progetto soddisfa tale requisito attraverso un meccanismo di persistenza basato su file di testo, gestito principalmente dai moduli `message_store.c` e `user_auth.c`.

## Scelte di Progetto

La scelta di utilizzare file di testo formattati in modo semplice (key: value) è stata dettata dalla necessità di semplicità e trasparenza, mantenendo il progetto leggero e facilmente portabile.

- **File degli Utenti (`data/users.txt`):** Gestito da `user_auth.c`, questo file memorizza le credenziali degli utenti. Ogni riga contiene un nome utente e la versione "hashed" della sua password, separati da uno spazio.
- **File dei Messaggi (data/messages.txt):** Gestito da `message_store.c`, questo file contiene tutti i messaggi della bacheca. Ogni messaggio è rappresentato da un blocco di testo che include `ID, autore, timestamp, oggetto e corpo,` delimitato da una riga `===END===`.

## Meccanismi di Caricamento e Salvataggio

**1. Caricamento all'Avvio (Loading):**
All'avvio del server, prima di iniziare ad accettare connessioni, viene inizializzato il `message_store`. La funzione `message_store_init` (in `message_store.c`) viene chiamata dal main del server. Questa funzione, a sua volta, invoca `load_messages`, che si occupa di:

- Aprire il file `data/messages.txt` in modalità lettura.
- Eseguire il parsing del file riga per riga, ricostruendo la struttura dati di ogni messaggio in memoria. I messaggi vengono caricati in un array dinamico (`MessageArray`).
- Durante il parsing, viene tenuta traccia dell'ID più alto per inizializzare correttamente il contatore `next_id`, garantendo che i nuovi messaggi abbiano sempre un ID univoco.

**2. Salvataggio alla Chiusura (Saving):**
La robustezza del sistema è garantita da come gestisce la chiusura. Il salvataggio dei dati non

avviene a ogni modifica (per motivi di performance), ma solo quando il server viene spento correttamente. Questo è orchestrato da due meccanismi principali in `server.c`:

- **Handler di Segnale (`handle_sigint`):** Il server imposta un gestore per il segnale `SIGINT`. Quando il segnale viene ricevuto, l'handler non termina immediatamente il processo: se ci sono client attivi, attende la loro disconnessione; mentre se non ci sono avvia lo shutdown del programma.
- **Funzione di Cleanup (`atexit`):** Nel main del server, la funzione `cleanup` viene registrata tramite `atexit(cleanup)`. Questa funzione C standard garantisce che cleanup venga eseguita automaticamente quando il programma termina normalmente. All'interno di cleanup, viene invocata `message_store_shutdown`, che a sua volta chiama `save_messages`. La funzione `save_messages` (in `message_store.c`) itera attraverso l'array di messaggi in memoria e li scrive nel file `data/messages.txt`, sovrascrivendo la versione precedente con i dati aggiornati.

---

# ID MESSAGGIO

In un sistema di bacheca, è essenziale che ogni messaggio possa essere identificato in modo univoco. Questo è cruciale per operazioni come la visualizzazione e, soprattutto, la cancellazione. Questo progetto garantisce l'unicità tramite un identificatore numerico (`ID`) intero e senza segno.

## Implementazione dell'ID Univoco

La gestione degli `ID` è interamente confinata nel modulo `message_store.c`. La struttura `MessageArray` contiene un campo `uint32_t next_id`, che funziona da contatore globale per il prossimo `ID` da assegnare.

**1. Generazione dell'`ID`:**
Quando un utente pubblica un nuovo messaggio, la funzione `add_message` viene invocata sul server. Al suo interno:

- Viene allocato spazio per un nuovo messaggio nell'array dinamico.
- L'`ID` del nuovo messaggio viene impostato al valore corrente di `message_array.next_id`.
- Subito dopo, `next_id` viene incrementato (`message_array.next_id++`).

Questo meccanismo, protetto da un mutex (`pthread_mutex_lock(&message_array.mutex)`), garantisce che anche con più thread che tentano di aggiungere messaggi contemporaneamente, ogni messaggio riceverà un **`ID` sequenziale e univoco**.

**2. Persistenza dell'`ID` e Gestione dei Riavvi:**
Il problema principale della persistenza e unicità dell'`ID` si presenta ad un nuovo start up del programma. Se `next_id` ripartisse sempre da 1, si potrebbero generare `ID` duplicati rispetto a quelli già presenti nel file di persistenza.

Il sistema risolve questo problema durante la fase di caricamento. La funzione `load_messages`:

- Inizializza una variabile locale `max_id` a 0.

- Durante la lettura di ogni messaggio dal file, confronta l'ID del messaggio letto (`current_msg.id`) con `max_id`. Se l'`ID` letto è **maggiore**, `max_id` viene **aggiornato**.
- Al termine della lettura di tutti i messaggi, il contatore globale viene inizializzato a `max_id + 1`.

Questo assicura che, anche dopo un riavvio, il server non **riutilizzerà mai un `ID`**, preservando l'unicità in modo perpetuo per tutta la vita dell'applicazione. Il tipo `uint32_t` offre un range di oltre 4 miliardi di `ID`, ampiamente sufficiente per lo scopo del progetto.

---

# STANDARDIZZAZIONE DELLO SCAMBIO DATI CLIENT-SERVER

In un'applicazione client-server, è fondamentale che entrambe le parti concordino sulle dimensioni massime dei dati che possono essere scambiati (es. lunghezza di username, password, messaggi) e sui codici che definiscono il protocollo di comunicazione. Una discrepanza in queste definizioni può portare a bug gravi, come buffer overflow, troncamento di dati o errori di interpretazione del protocollo.

## Centralizzazione delle Definizioni

Per affrontare questo problema, il progetto adotta una pratica di sviluppo solida: centralizzare tutte le definizioni comuni in file header condivisi, inclusi sia nel client che nel server.

**1. `common/common.h`:**
In common vengono definite le costanti che caratterizzano le dimensioni massime dei dati.

```
#define MAX_USERNAME_LEN 50

#define MAX_PASSWORD_LEN 50
#define MAX_SUBJECT_LEN  128
#define MAX_BODY_LEN     1024
```

Lato Client: Questi valori sono usati per dichiarare i buffer che conterranno l'input dell'utente. Ad esempio, in `client_api.c`, le variabili `username` e `password` sono dichiarate con queste dimensioni: `char username[MAX_USERNAME_LEN];`. Questo previene che l'utente inserisca dati più grandi di quanto il server si aspetti.
Lato Server: Sul server, queste costanti sono usate per definire le dimensioni dei campi nelle strutture dati (es. `char author[MAX_USERNAME_LEN]` nella `struct Message` in `message_store.c`) e dei buffer temporanei per la ricezione dei dati.

L'uso di queste costanti condivise garantisce coerenza e previene vulnerabilità legate alla gestione dei buffer.

**2. `common/protocol.h`:**
Questo file definisce la struttura del protocollo di comunicazione binario, che è molto più efficiente e meno ambiguo di un protocollo testuale.

`command_type e status_code (enums)`: Definiscono in modo non ambiguo tutti i possibili comandi che il client può inviare (`C_REGISTER, C_POST_MESSAGE`, etc.) e tutti i possibili codici di

stato che il server può restituire (`OK, AUTH_SUCCESS, ERROR`, etc.). L'uso di `enum` rende il codice più leggibile e manutenibile rispetto all'uso di costanti intere.
`packet_header` (struct): Definisce la struttura di ogni pacchetto inviato sulla rete.

```
typedef struct {
    uint8_t type;    // Tipo di comando o di stato
    uint32_t length; // Lunghezza del payload che segue
} packet_header;
```

Questo header a dimensione fissa precede ogni comunicazione. Il client e il server leggono prima l'header(5 byte). Il campo `type` dice loro cosa aspettarsi, e il campo `length` dice loro quanti byte aggiuntivi leggere per il payload (se presente). Questo approccio è robusto perché evita di dover fare il parsing di stringhe per determinare la fine di un messaggio e gestisce correttamente l'invio di dati binari o di testo contenente caratteri speciali.

L'adozione di questi file header condivisi è una scelta di progettazione cruciale che aumenta la robustezza, la sicurezza e la manutenibilità dell'intero sistema, perché consente di definire un protocollo globale adottato dall'intera architettura.

---

# AUTENTICAZIONE/ISCRIZIONE AL SISTEMA

Un sistema multiutente richiede un meccanismo sicuro per gestire l'identità degli utenti. Questo progetto implementa le due operazioni fondamentali: registrazione (iscrizione) e autenticazione (login). La logica è implementata nel modulo `user_auth.c` e gestita da `client_handler.c` sul server e `client_api.c` sul client.

## Flusso della Registrazione

1. **Client:** L'utente seleziona l'opzione "Registrati". La funzione `c_register` (in `client_api.c`) chiama `get_credentials` per ottenere username e password.
2. **Payload:** Le credenziali vengono assemblate in un unico payload. L'username (stringa terminata da \0) è seguito immediatamente dalla password (anch'essa terminata da \0). Questo è un modo efficiente per inviare due stringhe in un unico buffer.
3. **Invio:** Il client invia un pacchetto al server con `header.type = C_REGISTER` e il payload contenente le credenziali.
4. **Server:** Il `client_handler` riceve il pacchetto. Identifica il tipo `C_REGISTER`, estrae username e password dal payload e invoca `register_user(user, pass)`.
5. **Logica di Registrazione (`register_user`):**
    - Acquisisce un mutex (`pthread_mutex_lock(&user_mutex)`) per garantire che non ci siano accessi concorrenti al file `users.txt`.
    - Controlla se l'utente esiste già leggendo il file.
    - Se l'utente non esiste, esegue l'hash della password (vedi sezione successiva) e aggiunge la nuova coppia username hashed_password al file.
    - Rilascia il mutex e restituisce `true` (successo) o `false` (utente già esistente).
6. **Risposta:** Il server invia un pacchetto di stato al client: `REG_SUCCESS` o `REG_USER_EXISTS`.
7. **Client:** La funzione `wait_for_status` riceve la risposta e il client informa l'utente dell'esito, comunicando in caso di insuccesso il codice di errore corrispondente.

### Flusso dell'Autenticazione

Il flusso del login è molto simile a quello della registrazione.

1. **Client:** L'utente seleziona "Accedi". La funzione `c_login` raccoglie le credenziali tramite `get_credentials`.
2. **Invio:** Viene inviato un pacchetto con `header.type = C_LOGIN` e lo stesso formato di payload.
3. **Server:** Il `client_handler` riceve la richiesta e invoca `authenticate_user(user, pass)`.
4. **Logica di Autenticazione (`authenticate_user`):**
    - Acquisisce il mutex `user_mutex`.
    - Esegue l'hash della password fornita.
    - Legge il file `users.txt` riga per riga, confrontando l'username e l'hash della password con ogni record presente.
    - Se viene trovata una corrispondenza, rilascia il mutex e restituisce `true`. Altrimenti, `false`.
5. **Gestione dello Stato:** Se l'autenticazione ha successo, il thread `client_handler` imposta una variabile booleana `auth = true` e memorizza il nome dell'utente in `curr_user`. Questo stato di "login effettuato" è mantenuto per tutta la durata della connessione del client e viene usato per autorizzare le operazioni successive (come postare o cancellare messaggi).
6. **Risposta:** Il server risponde con `AUTH_SUCCESS o AUTH_FAILURE`.
7. **Client:** Se riceve `AUTH_SUCCESS`, il client imposta la sua variabile `b_log = true`, che modifica il menu mostrato all'utente, sbloccando le funzionalità della bacheca.

Questa architettura separa nettamente l'interfaccia utente (sul client), la comunicazione di rete (API client e protocollo) e la logica di business (sul server), rendendo il sistema modulare e più facile da comprendere.

---

# SICUREZZA DEI DATI E DEL SISTEMA

La sicurezza è un aspetto critico, specialmente per quanto riguarda la gestione delle credenziali degli utenti. Sebbene il progetto implementi alcune misure di base, è importante analizzarne i punti di forza.

### Hashing delle Password

Il sistema **non memorizza mai le password in chiaro** sul disco. Questo è un principio fondamentale della sicurezza. Invece, utilizza una funzione di hash.

**Implementazione (`user_auth.c`):**
La funzione `hash_password` implementa l'algoritmo **djb2**, un hash non crittografico molto semplice e veloce.

**Analisi della Sicurezza:**

- **Punto di Forza:** L'uso di un hash è meglio che memorizzare password in chiaro. Se il file `users.txt` venisse compromesso, un aggressore non otterrebbe direttamente le password degli utenti.

- **Punto di Debolezza Critico:** L'algoritmo djb2 **non è un hash crittografico sicuro**. È stato progettato per le hash table, non per la sicurezza. I suoi principali difetti in questo contesto sono:
    1. Velocità di calcolo
    2. Mancanza di Randomicità

### Protezione da Buffer Overflow

Il progetto presta attenzione a prevenire buffer overflow utilizzando le costanti definite in `common.h` e funzioni di libreria sicure dove possibile. Questo riduce il rischio di vulnerabilità legate alla corruzione della memoria.

---

# GESTIONE DEI SEGNALI

La gestione dei segnali è un aspetto fondamentale per la robustezza di applicazioni a lunga esecuzione come un server o un client interattivo.

### Gestione dei Segnali sul Server

L'obiettivo principale del server è garantire una shutdown, ovvero una chiusura pulita che permetta di salvare lo stato corrente e non interrompa bruscamente le operazioni dei client.

**Implementazione (server.c):**

1. **Blocco dei Segnali per i Thread:** All'inizio del main, viene utilizzata `pthread_sigmask` per bloccare quasi tutti i segnali per il thread principale e, per ereditarietà, per tutti i thread che verranno creati. Viene lasciato sbloccato solo SIGINT. Questo assicura che solo il thread principale gestisca i segnali, evitando comportamenti complessi e imprevedibili se più thread reagissero allo stesso segnale.
2. **Handler per SIGINT:** Viene installato un handler, `handle_sigint`, per il segnale SIGINT (Ctrl+C) utilizzando `sigaction`.
3. **Logica dell'Handler:** Quando `handle_sigint` viene eseguito, non termina il server. Invece:
    - Controlla il contatore globale `active_client_count` (protetto da mutex).
    - Se non ci sono client attivi, permette al programma di terminare, il che attiverà la funzione di `cleanup` registrata con `atexit`.
    - Se ci sono client attivi, stampa un messaggio di attesa e non fa nulla, permettendo al server di continuare a servire i client esistenti. L'idea è che l'amministratore possa attendere la disconnessione naturale o inviare un secondo SIGINT se necessario.

Questa strategia bilancia la necessità di chiudere il server con quella di non interrompere il servizio per i client connessi.

### Gestione dei Segnali sul Client

Sul client, l'obiettivo è diverso: prevenire una chiusura accidentale da parte dell'utente.

**Implementazione (client.c):**
Nel main del client, i segnali SIGINT (Ctrl+C) e SIGTSTP (Ctrl+Z) vengono ignorati impostando il loro handler a SIG_IGN tramite `sigaction`.

Questo migliora l'esperienza utente: se l'utente preme accidentalmente Ctrl+C mentre sta scrivendo un messaggio, l'applicazione non si chiuderà. L'unico modo per terminare il programma è scegliere esplicitamente l'opzione "Esci" dal menu, che fa terminare il main_loop in modo controllato.

---

# GESTIONE DELLE DISCONESSIONI

## Rilevamento della Disconnessione

La gestione di ogni client è affidata a un thread separato che esegue la funzione `handle_client`. Questa funzione contiene un ciclo while che attende e processa i comandi dal client. Il cuore del rilevamento della disconnessione si trova nella chiamata a `recv_all`.

```
    // In client_handler.c
while (recv_all(sock, &header, sizeof(header)) == 0) {
    // ... processa il pacchetto ...
}
```

La funzione `recv` ha due possibili valori di ritorno:

- Se `recv` restituisce 0, significa che il client ha chiuso la connessione in modo ordinato.
- Se `recv` restituisce -1, si è verificato un errore. Questo spesso indica una disconnessione forzata (es. la connessione è stata resettata).

In entrambi i casi, `recv_all` è progettato per restituire -1 per segnalare la fine della comunicazione. Quando questo accade, la condizione del ciclo while in `handle_client` diventa falsa e il ciclo termina.

## Pulizia delle Risorse

Una volta che il ciclo termina, l'esecuzione del thread `handle_client` prosegue verso la fine della funzione, dove avviene la pulizia delle risorse associate a quel client:

1. **Chiusura del Socket:** La prima operazione è `close(sock)`. Questo rilascia il file descriptor del socket, liberando le risorse associate a livello di sistema operativo.
2. **Decremento del Contatore Clienti:** decrementa il contatore globale `active_client_count` in maniera thread-safe, ovvero bloccando con `pthread_mutex_lock` e una volta decrementato sbloccando il mutex `pthread_mutex_unlock`.

Questo meccanismo è robusto ed essenziale per la stabilità a lungo termine del server. Senza di esso, ogni disconnessione di un client lascerebbe un thread e un socket aperti, portando rapidamente all'esaurimento delle risorse del server. La gestione corretta della disconnessione assicura che il server possa funzionare indefinitamente, gestendo un numero arbitrario di connessioni nel tempo (limitate solo dal numero massimo di thread concorrenti definiti nel thread pool).

---

# API DELLA BACHECA

Le funzioni principali della bacheca (visualizzazione, invio e cancellazione di messaggi) sono implementate attraverso un protocollo di richiesta-risposta ben definito.

## Visualizzazione della Bacheca (`c_get_board`)

Questa è l'interazione più complessa, poiché può richiedere l'invio di più pacchetti dal server al client.

1. **Client:** L'utente sceglie "Visualizza messaggi". La funzione `c_get_board` invia un semplice pacchetto di comando al server con tipo C_GET_BOARD e lunghezza 0.
2. **Server (`get_board in message_store.c`):**
    - Acquisisce il mutex della bacheca per garantire una lettura consistente.
    - **Ordinamento:** I messaggi vengono ordinati cronologicamente. Il codice fornito implementa un bubble sort basato sulla conversione del `timestamp` (stringa) in un oggetto `time_t`.
    - **Invio Multi-pacchetto:** Invece di inviare un unico, enorme pacchetto con tutta la bacheca, il server itera sui messaggi ordinati e invia ogni messaggio (o gruppo di messaggi, come le intestazioni di data) in un pacchetto separato con tipo OK. Questo approccio è più flessibile e gestisce meglio la memoria, evitando di dover allocare un buffer gigantesco.
    - **Segnale di Fine:** Al termine dell'invio di tutti i messaggi, il server invia un ultimo pacchetto speciale con tipo END_BOARD.
3. **Client (`c_get_board`):**
    - Entra in un ciclo while(1).
    - In ogni iterazione, attende di ricevere un packet_header dal server.
    - Se `header.type == END_BOARD`, il ciclo si interrompe.
    - Se `header.type == OK`, legge il payload della dimensione specificata in `header.length` e lo stampa sullo schermo.
    - Se riceve un tipo inaspettato, segnala un errore.

Questo protocollo di streaming è efficiente e scalabile.

## Invio di un Messaggio (`c_post_message`)

1. **Client:** L'utente sceglie "Invia un messaggio". `c_post_message` chiama `get_content` per raccogliere l'oggetto e il corpo del messaggio dall'utente.
2. **Payload:** Similmente a login/registrazione, oggetto e corpo vengono concatenati in un unico payload, separati da un terminatore nullo.
3. **Invio:** Viene inviato un pacchetto con tipo C_POST_MESSAGE e il payload.
4. **Server (`handle_client`):**
    - Riceve il pacchetto.
    - Estrae oggetto e corpo dal payload.
    - Chiama `add_message(curr_user, subject, body)`, che aggiunge il messaggio all'archivio in memoria (in modo thread-safe).
    - Risponde al client con un pacchetto di stato OK.
5. **Client:** `wait_for_status` attende la conferma OK e informa l'utente del successo.

### Cancellazione di un Messaggio (`c_delete_message`)

1. **Client:** L'utente sceglie "Cancella un messaggio". `c_delete_message` chiede all'utente di inserire l'ID del messaggio da cancellare.
2. **Payload:** L'ID viene usato direttamente come payload.
3. **Invio:** Viene inviato un pacchetto con tipo C_DELETE_MESSAGE e un payload di 4 byte (la dimensione di `uint32_t`).
4. **Server (`handle_client`):**
    - Riceve il pacchetto e verifica che la lunghezza sia esattamente 4 byte.
    - Estrae l'ID del messaggio dal payload.
    - Chiama `delete_message(message_id, curr_user)`.
5. **Logica di Cancellazione (`delete_message`):**
    - **Autorizzazione:** Questa funzione non si limita a cancellare. Prima cerca il messaggio con l'ID specificato. Una volta trovato, **verifica che il `current_user` (l'utente che ha fatto la richiesta) sia lo stesso dell'autore del messaggio**. Questo è un controllo di sicurezza cruciale.
    - **Cancellazione:** Se l'utente è autorizzato, il messaggio viene rimosso dall'array in memoria (vedi sezione successiva).
    - **Risposta:** La funzione restituisce un codice che indica successo, fallimento (messaggio non trovato) o non autorizzato.
6. **Risposta del Server:** In base al risultato di `delete_message`, il server invia uno stato OK, NOT_FOUND o UNAUTHORIZED.
7. **Client:** `wait_for_status` interpreta la risposta e informa l'utente.

---

# CANCELLAZIONE DI UN MESSAGGIO, SICUREZZA ED EFFICIENZA

L'operazione di cancellazione di un messaggio merita un'analisi più approfondita, sia per la logica di autorizzazione che per l'efficienza dell'implementazione.

### Logica di Autorizzazione

Come menzionato nella parte precedente, il controllo di autorizzazione è un punto chiave. È implementato in `delete_message` (in `message_store.c`):

```
if (strcmp(message_array.messages[i].author, current_user) != 0) {
    pthread_mutex_unlock(&message_array.mutex);
    return -1; // Codice per "Non autorizzato"
}
```

Questa verifica assicura che un utente non possa cancellare i messaggi di altri, un requisito fondamentale per l'integrità della bacheca. La logica di autorizzazione risiede interamente sul server. Il client non ha alcun ruolo nel decidere se un'operazione è permessa; si limita a inviare la richiesta e a mostrare il risultato.

### Implementazione della Cancellazione e Analisi di Efficienza

I messaggi sono memorizzati in un array dinamico (`Message* messages`). La cancellazione di un elemento da un array richiede una risistemazione degli elementi successivi.

**L'algoritmo implementato in delete_message è il seguente:**

1. **Ricerca (O(N)):** Viene eseguita una scansione lineare dell'array per trovare l'indice (`found_index`) del messaggio con l'ID specificato. La complessità di questa operazione è, nel caso peggiore, proporzionale al numero di messaggi (N).
2. **Cancellazione e Shift (O(N)):** Una volta trovato l'elemento all'indice `found_index`, le sue risorse allocate dinamicamente (body e timestamp) vengono liberate con `free()`. Successivamente, tutti gli elementi successivi a `found_index` vengono spostati una posizione indietro per compattare l'array.

Questo secondo ciclo for ha anch'esso una complessità, nel caso peggiore (cancellazione del primo elemento), di O(N).

**Discussione delle Scelte Realizzative:**

- **Pro:** Per un numero di messaggi relativamente piccolo (migliaia o decine di migliaia), una complessità O(N) è perfettamente accettabile. L'implementazione è semplice, facile da capire e da mantenere. L'uso di un array dinamico consente un accesso in lettura molto veloce (O(1) per indice), il che è un vantaggio per l'operazione di visualizzazione della bacheca.
- **Contro:** Se il sistema dovesse scalare per gestire milioni di messaggi, un'operazione di cancellazione O(N) non può essere supportata.

Per il contesto del progetto, la scelta di un array dinamico rappresenta un ottimo compromesso tra semplicità implementativa e performance adeguate.

---

# GESTIONE STREAM TCP

TCP è un protocollo di trasporto orientato allo stream, affidabile e orientato alla connessione. Tuttavia, "affidabile" si riferisce al fatto che i dati arrivano senza errori e in ordine, ma **non garantisce che i dati inviati con una singola chiamata send() vengano ricevuti con una singola chiamata recv()**.

## Le Funzioni `send_all e recv_all`

Per affrontare questo problema, il progetto implementa due funzioni helper cruciali in common/net_utils.c: send_all e recv_all. Queste funzioni "wrappano" le chiamate di sistema send e recv in un ciclo per garantire che l'intera quantità di dati specificata venga trasferita.

**`send_all:`**

```
int send_all(int sockfd, const void *buf, size_t len){
    size_t c = 0;
    while(c < len){
        ssize_t sent = send(sockfd, (char*)buf + c, len - c, MSG_NOSIGNAL);
        if (sent <= 0) return -1;
```

```
        c += sent;
    }
    return 0;
}
```

- La funzione entra in un ciclo while che continua finché il numero di byte inviati (c) è inferiore alla lunghezza totale (len).
- In ogni iterazione, chiama `send` per tentare di inviare i byte rimanenti. Il puntatore al buffer viene incrementato (`(char*)buf + c`) e la lunghezza da inviare viene diminuita (`len - c`).
- Il ciclo termina solo quando tutti i byte sono stati inviati con successo o se send restituisce un errore.
- MSG_NOSIGNAL è usato per prevenire che il programma termini con un segnale SIGPIPE se si tenta di scrivere su un socket la cui connessione è già stata chiusa dal peer. Invece, send restituirà un errore (EPIPE), che la funzione gestisce restituendo -1.

**recv_all:**

```
int recv_all(int sockfd, void *buf, size_t len){
    size_t c = 0;
    while(c < len){
        ssize_t received = recv(sockfd, (char*)buf + c, len - c, 0);
        if (received <= 0) return -1;
        c += received;
    }
    return 0;
}
```

- Questa funzione è la controparte di `send_all` e funziona in modo speculare.
- Entra in un ciclo che chiama `recv` ripetutamente finché non ha ricevuto esattamente len byte.
- Se `recv` restituisce 0 o un valore negativo, indica che la connessione è stata chiusa o si è verificato un errore, e la funzione termina con un errore.

L'uso sistematico di `send_all` e `recv_all` in tutto il progetto (ad esempio, nelle funzioni `response` e in `client_handler`) è ciò che rende la comunicazione di rete robusta e affidabile a livello applicativo. Senza queste funzioni, il protocollo basato su `packet_header` fallirebbe, poiché non ci sarebbe alcuna garanzia di leggere un intero `header` o un intero payload in una sola volta. Questa è una delle scelte di progettazione più importanti per la correttezza del sistema.

---

# FUTURE WORKS

Il progetto nonostante abbia una base solida e ottimizzata che consente una user-experience accettabile, presenta diversi spunti per future modifiche, sia per quanto riguarda possibili nuove implementazioni, ma soprattutto sottili aggiornamenti che consentirebbero al progetto di convergere verso le versioni già implementate sul web.

Le principali modifiche riguardano soprattutto le modalità di gestione dei file, sia su disco che in memoria di lavoro:

- Suddivisione dei messaggi in più file di testo e creazioni di folder in base al timestamp dei messaggi; consentirebbe l'archiviazione di più messaggi nella bacheca elettronica, poiché adesso il limite superiore è data dalla dimensione massima di archiviazione dei file di testo nei vari SO
- Possibilità di richiedere di vedere solo specifici periodi di messaggi della bacheca; limita i tempi di attesa per un client che è interessato ad un periodo temporale e non all'interezza del contenuto della bacheca
- Maggiore sfruttamento della località per i messaggi presenti in RAM, senza la necessità di caricare ogni volta all'avvio del file tutta la bacheca di messaggi.

Oltre a nuovi metodi di gestione dell'archiviazione e di presenza dei messaggi in RAM durante l'esecuzione dell'applicativo, è possibile anche implementare misure di sicurezza maggiori sia per quanto riguarda le modalità di archiviazione sul disco, ma soprattutto per quanto riguarda la trasmissione tra client-server. Infatti, oltre ad effettuare l'hash della password prima di inserirlo nel file user.txt, anch'esso da migliorare con funzioni hash più sicure, si potrebbe implementare la criptazione e decriptazione dei messaggi in trasmissione tra client e server così da garantire uno scambio sicuro.

---

# CONCLUSIONI

In questa relazione ho descritto una bacheca elettronica implementata per gestire relazioni client-server in maniera consistente, ottimizzata e sicura. Il progetto mostra principi di sicurezza implementati in modo coerente, come l'hashing delle password, una UI intuitiva e facile da leggere, una gestione dei dati consistente, resiliente e duratura…

In generale, nonostante il software non sia pronto per essere distribuito, si è dimostrato essere una buona base per futuri sviluppi ed eventuali distribuzioni.

Gli obiettivi stabiliti sono stati conseguiti anche attraverso l'integrazione di funzionalità aggiuntive, introdotte con finalità sperimentali. Questa scelta ha permesso di valutare in modo più approfondito la flessibilità del sistema e le sue potenzialità evolutive.

Il progetto, tuttavia, offre ancora ampie possibilità di miglioramento, in particolare per quanto riguarda la scalabilità, il rafforzamento delle misure di sicurezza e l'ottimizzazione delle prestazioni. Questi aspetti rappresentano linee di sviluppo fondamentali per consolidare la solidità della piattaforma e garantirne l'affidabilità in contesti applicativi reali.