UNIVERSITÀ DEGLI STUDI DI NAPOLI "FEDERICO II"

DIETI - Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Progetto per Laboratorio di Sistemi Operativi
A.A. 2023/2024

Traduttore Client - Server

Autori:
Antonio Legnante - N86003588
Leo Cucurachi - N86004289

INDICE

1. Introduzione **2**
2. Struttura del sistema **3**
2.1 Descrizione Generale 3
2.2 Componenti e funzionalità principali 3
3. Struttura del Server **4**
3.1 Descrizione Generale 4
3.2 Gestione del parallelismo tra Client 4
3.3 Memoria condivisa tra threads e Race Conditions 5
- Room Mutex e Condition Variable 5
- Rooms Array Read-Write Mutex 5
- Client Array 6
3.4 Database 6
3.5 Librerie esterne 6
4. Client **7**
4.1 Descrizione Generale 7
4.2 Interazione con il Server 7
4.3 Librerie esterne 7
5. Docker Compose **7**
5.1 Esecuzione con Docker Compose 7
5.2 Vantaggi di Docker Compose 8

1. Introduzione

Il progetto consiste nello sviluppo di un'applicazione per la gestione di una chat divisa a stanze per un numero non precisato di utenti.
L'obiettivo del progetto è creare un sistema di chat multilingue che faciliti la comunicazione tra utenti di diverse lingue.
Le funzionalità principali sono:

- Registrazione degli utenti con scelta della lingua.
- Autenticazione tramite login.
- Accesso alle stanze di chat in base alla lingua.
- Traduzione automatica dei messaggi nella lingua specifica della stanza.
- Creazione di nuove stanze da parte degli utenti.

2. Struttura del sistema

2.1 Descrizione Generale

Il sistema è composto da un Client ed un Server, entrambi scritti in C, che comunicano tramite socket. Le richieste dei Client vengono servite dal Server, che si appoggia su un Database SQLite per mantenere la persistenza di alcune informazioni.

2.2 Componenti e funzionalità principali

1. **Client**:
    - Registrazione degli utenti.
    - Autenticazione tramite login.
    - Invio e ricezione di messaggi.
    - Interfaccia utente sul terminale per la selezione delle stanze e la visualizzazione dei messaggi tradotti.
2. **Server**:
    - Autenticazione degli utenti.
    - Gestione delle stanze di chat.
    - Traduzione dei messaggi inviati dagli utenti.
    - Monitoraggio e gestione degli utenti inattivi.
    - Comunicazione con il database per la gestione degli utenti e del dizionario di traduzione.
    - Creazione di nuovi thread per gestire le connessioni dei client.

3. **Database:**
    - Memorizzare le informazioni relative agli utenti (registrazione e autenticazione).
    - Memorizzare il dizionario di traduzione delle parole.
    - Assicurare la persistenza dei dati tra le sessioni del server.

3. Struttura del Server

3.1 Descrizione Generale

Il Server è scritto in linguaggio C. L'accettazione dei Client avviene mendiante l'utilizzo di una Socket TCP. All'accettazione del nuovo Client viene creata una socket specifica per il Client, utilizzata per tutte le comunicazioni tra Client e Server, come Registrazione, Login, invio e ricezione di Messaggi. Il Server per la gestione in parallelo delle richieste dei Client, crea un nuovo thread per ogni nuovo Client accettato.

3.2 Gestione del parallelismo tra Client

Per la gestione in parallelo di più client, il Server crea un nuovo thread, specifico per il Client, ad ogni nuova connessione. Questo approccio ha tre vantaggi principali:

1. **Concorrenza e Parallelismo**: Ogni Client è gestito in un thread separato, permettendo al server di gestire più Client contemporaneamente senza bloccare le altre connessioni. Questo migliora la reattività e la capacità del server di gestire molteplici richieste simultaneamente.
2. **Isolamento delle Connessioni**: Utilizzando thread separati, ogni connessione Client è isolata dalle altre. Questo significa che un problema o un crash in un thread non influenzerà le altre connessioni, migliorando la robustezza e la stabilità complessiva del server.
3. **Facilità di Implementazione del Timeout**: Con thread separati, è più facile implementare meccanismi di timeout per disconnettere i client inattivi. Ogni thread può monitorare il tempo di inattività del proprio client e chiudere la connessione se supera una certa soglia.

3.3 Memoria condivisa tra threads e Race Conditions

- Room Mutex e Condition Variable

Il server mantiene una struttura dati globale condivisa da tutti i thread: l'array contenente tutti i riferimenti alle stanze. Le singole stanze sono implementate come Struct contenenti varie informazioni, tra cui la lista dei client connessi e la coda dei client in attesa di entrare nella stanza. Le operazioni di invio di un messaggio a tutti i Client connessi in una stanza, e la gestione della coda di attesa, potrebbero causare race condition se non correttamente gestite. Per questo motivo, ogni stanza possiede un mutex e una variabile condizionale. Prima di effettuare una qualsiasi operazione in una stanza, come l'uscita, l'ingresso, l'invio di un messaggio ecc. un Client deve prima acquisire il mutex di quella stanza. La variabile condizionale serve per porre in attesa i Client che si mettono in coda per entrare nella stanza, e garantire che l'ordine di uscita dalla coda di attesa sia di tipo FIFO. Tale operazione, infatti, richiede dei controlli per garantire che il thread risvegliato dall'attesa, sia effettivamente quello in cima alla coda di attesa, visto che, citando il manuale di pthreads:

> *If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked.*

Questo approccio garantisce l'indipendenza e la possibilità di operazioni in parallelo su stanze differenti, e l'atomicità delle operazioni su una stessa stanza da parte di due Client, oltre che la gestione FIFO della coda di attesa.

- Rooms Array Read-Write Mutex

Un secondo problema di race condition si ha con l'operazione di aggiunta di una stanza. Il server, infatti, mantiene un array di riferimenti alle stanze, caricate all'avvio del server dal database (inizializzato con una dimensione pari a 1.5 * il numero di stanze), riallocando dinamicamente la memoria dopo le aggiunte, quando l'array risulta pieno. Essendo l'aggiunta un'operazione meno frequente rispetto all'invio di messaggi, all'ingresso e all'uscita da una stanza, risulta più utile l'accesso diretto ad una stanza specifica garantito dalla struttura dell'array, piuttosto che la maggiore flessibilità nell'aggiunta di stanze (più semplice, ad esempio, utilizzando una lista). Quando un Client vuole aggiungere una stanza, deve acquisire un rdwr_lock globale in modalità di

scrittura. Tale lock deve essere acquisito, invece, in modalità di lettura dagli altri Client che vogliono salvare il riferimento ad una specifica stanza (nel momento dell'accesso alla stessa), per poi inviarvi messaggi. In questo modo tutti i Client che vorranno entrare all'interno di una stanza, copiandone il riferimento, dovranno attendere che un Client finisca eventualmente di aggiungere una nuova stanza, di modo da garantire la coerenza dei riferimenti. Non potendoci essere eliminazioni di stanze, è garantito che nel momento in cui un Client acquisisca un riferimento ad una stanza, tale riferimento rimanga invariato.

- Client Array

Il server mantiene anche un arrray di Client globale, ma tale array non dà luogo a race condition. L'unico thread che si occupa di istanziare le struct per i nuovi Client, infatti, è il thread principale, mentre i singoli thread specifici per i Client hanno il riferimento specifico al proprio Client.

3.4 Database

Per il database si è scelto di utilizzare SQLite3 per memorizzare sia le informazioni relative agli utenti, utili per il login, che il dizionario per la traduzione dei messaggi.
La scelta di SQLite3 è data dai seguenti motivi:

1. **Semplicità di Integrazione**: SQLite3 è una libreria leggera che fornisce un database SQL auto-contenuto, senza bisogno di configurazioni server complesse. È facile da integrare direttamente nelle applicazioni C.
2. **Semplicità di Configurazione**: Non richiede un server separato per funzionare.
3. **Performance**: Per applicazioni di piccole e medie dimensioni, SQLite3 offre performance eccellenti con bassi tempi di risposta, essendo altamente ottimizzato per operazioni di lettura/scrittura su file locali.
4. **Richieste di Storage Minime**: SQLite3 occupa poco spazio su disco e memoria, rendendolo adatto anche per dispositivi con risorse limitate.

3.5 Librerie esterne

Il server dipende dalle seguenti librerie esterne:

1. sqlite3 per la connessione e la gestione del database SQLite.

2. sodium per il salting e l’hashing delle password da memorizzare all’interno del database

4. Client

4.1 Descrizione Generale

Il Client funziona completamente dalla shell, mantenendo un’interfaccia intuitiva verso l’utente. Questa scelta è stata fatta per allinearsi con le tipiche applicazioni del mondo Linux, permettendo l'esecuzione esclusivamente da terminale. Per la gestione della TUI (Text User Interface), è stata utilizzata una libreria esterna, **ncurses**, che offre funzionalità avanzate per una migliore gestione dell’interfaccia del terminale, come la navigazione dei menu e la gestione delle finestre.

4.2 Interazione con il Server

Il Client comunica con il server tramite Socket TCP. Dopo aver stabilito la connessione, utilizza questa socket per lo scambio di tutte le informazioni, tra cui:

- Invio delle credenziali per la registrazione e il login.
- Invio di messaggi alle stanze di chat.
- Ricezione dei messaggi da altri utenti nella stessa stanza.

Questa architettura permette una comunicazione efficiente e in tempo reale tra Client e Server.

4.3 Librerie esterne

Il Client dipende dalle seguenti librerie esterne:

1. ncurses per la migliore gestione della TUI.

5. Docker Compose

5.1 Esecuzione con Docker Compose

Per l’esecuzione del sistema in locale è possibile utilizzare Docker Compose. Sia il Client che il Server sono eseguibili mediante Docker Compose. Per

l’esecuzione del sistema basta eseguire i seguenti comandi nella Root Directory:

```
$ docker compose build
$ docker compose up server
$ docker compose run client
```

Il Client puo essere eseguito anche su più terminali interattivamente con il terzo comando.

5.2 Vantaggi di Docker Compose

I vantaggi nell’utilizzo di docker compose sono stati:

1. **Facilità di Setup e Deployment:** Docker Compose semplifica notevolmente il processo di setup e deployment dell'applicazione. Con un singolo comando è possibile avviare e configurare tutti i servizi necessari, eliminando la complessità di configurare manualmente ogni componente.
2. **Isolamento degli Ambienti:** Docker Compose garantisce che ogni servizio (client e server) venga eseguito in un container isolato. Questo isolamento previene conflitti tra dipendenze e ambienti, garantendo che l'applicazione funzioni correttamente indipendentemente dalla configurazione del sistema host.
3. **Portabilità:** Le configurazioni Docker Compose possono essere facilmente condivise e utilizzate su qualsiasi sistema che supporti Docker.
4. **Configurazione Automatizzata della Rete:** Docker Compose crea automaticamente una rete virtuale condivisa per tutti i container definiti nel file docker-compose.yml. Questo permette ai container di comunicare tra loro tramite nome del servizio, senza bisogno di configurare manualmente gli indirizzi IP.