MULTICHAT

Progetto di Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica
Anno Accademico 2022-2023

N86003280 Annunziata Valentina
N86003285 Ciccarelli Francesco
N86003446 De Fenza Stefano

INDICE

Descrizione Generale

Multichat è un applicativo Client-Server che permette agli utenti di creare stanze (o di accedervi) e scambiare messaggi tra i presenti al loro interno.
È possibile registrarsi ed accedere al sistema con il proprio account.
L'accesso alle stanze viene regolato da un utente "master", che può accettare o rifiutare l'ingresso.
I messaggi inviati non vengono immagazzinati in maniera persistente.
Per questo motivo i messaggi scompariranno al termine della conversazione, garantendo la privacy degli utenti.

Principali Funzionalità in breve

- Database per la gestione del login utenti
- Creazione e gestione di più stanze e degli utenti al loro interno
- Accettazione all'interno della stanza da parte del proprietario
- Passaggio della proprietà della stanza ad un altro utente
- Invio di messaggi a tutti gli utenti presenti nella stanza
- Invio di messaggi di avviso del server a tutti gli utenti in una stanza

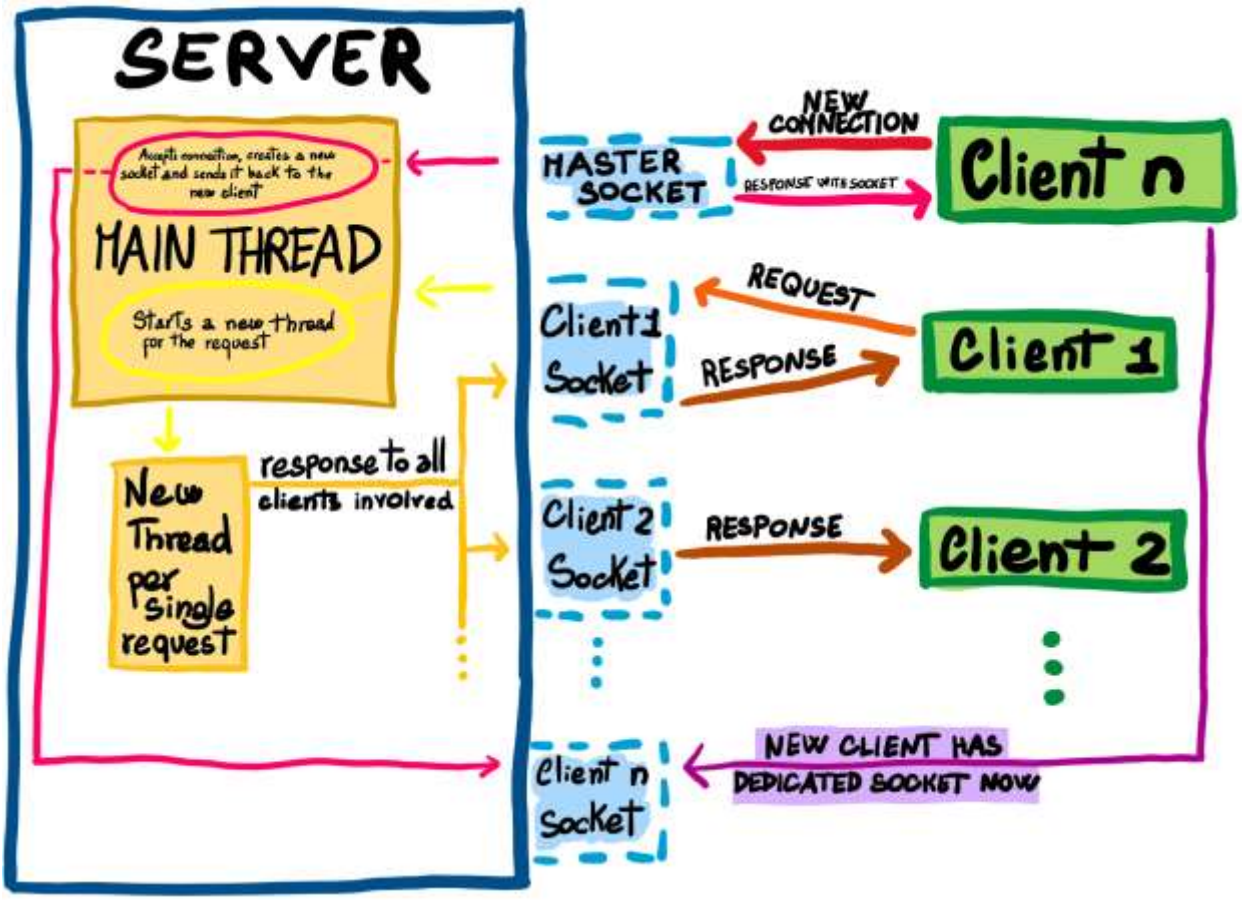
Server

Realizzato in linguaggio C ed eseguito in ambiente UNIX, è il sistema centrale che riceve e soddisfa le richieste dei client connessi, facendo da tramite e svolgendo le operazioni per fornire i servizi descritti nel paragrafo precedente.
La comunicazione avviene tramite Socket con il protocollo TCP.

Ogni Client, all'avvio, si connette al server tramite una socket principale. Il Server, quindi, crea per ogni Client una socket dedicata.

Viene utilizzata la funzionalità `select` per la gestione in arrivo delle richieste. Ciò ha evitato la definizione esplicita di Queue o altre strutture per la gestione delle richieste.

Inoltre, poiché si prevede l'arrivo di numerose richieste contemporaneamente da parte di diversi Client, il sistema utilizza il Multithreading, per soddisfarle in maniera parallela ed indipendente, ove possibile.



Il meccanismo di `select` viene utilizzato per monitorare più File Descriptor contemporaneamente e in maniera efficiente, poiché è offerto dal Sistema Operativo ed agisce a basso livello. La funzione è presente nella libreria C standard `<sys/select.h>` su sistemi basati su UNIX e POSIX.
All'interno del progetto i File Descriptor monitorati con questo meccanismo sono quelli delle socket create per la comunicazione con i Client.

`server.c`

In questo file è presente il main del programma, dove vengono gestite le socket e la creazione dei thread per soddisfare le richieste.

La chiamata a `select` mette in attesa il thread principale del server, finché non riceve una richiesta su una delle socket. Dopodiché

- Se la richiesta ricevuta proviene dalla socket principale, si tratta di un nuovo client appena connesso. Il Server, quindi, provvederà ad aprire una nuova socket per questo client, aggiungendo il suo File Descriptor tra quelli monitorati dalla `select`.
- Se la richiesta ricevuta proviene da un'altra socket, si tratta di un client già connesso che necessita di una funzionalità specifica del sistema.

Verrà creato un thread apposito che eseguirà la funzione `socket_dispatcher()`, all'interno del file `socket_handler.c` che si occupa di soddisfare le richieste specifiche dei client. Una volta soddisfatta la richiesta il thread viene distrutto.
Il sistema utilizza quindi il modello Thread-Per-Request.

socket_handler.c

In questo file è presente la logica di interpretazione delle richieste, tramite l'utilizzo di TAG di 5 byte inseriti ad inizio richiesta da parte dei client.
Ogni TAG corrisponde ad una specifica funzionalità:

TAG	Funzionamento
[LGN]username<>password	Permette ad un utente di accedere inviando username e password che verranno verificati per l'accesso.
[RGT]username<>password	Permette ad un utente di registrarsi inviando username e password che verranno salvati.
[CRT]RoomName	Permette di creare una stanza, scegliendo un nome che sarà visibile agli utenti.
[MSG]message<>room_id	Invia il messaggio a tutti i membri della stanza specificata attraverso l'id.
[RQT]room_id	Invia una richiesta di accesso al master della stanza.
[ACC]socket_id<>room_id	Permette al master di accettare la richiesta di accesso fatta da un utente.
[NAC]socket_id_client	Permette al master di rifiutare la richiesta di accesso fatta da un utente.
[EXT]room_id	Permette di uscire dalla stanza specificando l'id.
[LST]	Permette di ricevere la lista di tutte le stanze create dagli utenti.

In questo file è presente anche la logica di risposta per il client attraverso la scrittura all'interno della corrispettiva socket. La gestione degli utenti e delle stanze avviene invece all'interno del file `rooms_handler.c`.

Struttura e manipolazione dei dati

Si è scelto di utilizzare un approccio simile a quello orientato agli oggetti, definendo delle struct - `Room`, `Client` ed `User` – all'interno di omonimi file, nei quali sono definite anche funzioni utili alla allocazione, manipolazione e distruzione delle stesse. Le funzioni che interagiscono con queste strutture lo fanno sempre attraverso puntatori alle stesse.

rooms_handler.c

In questo file è definita la struttura principale del server: un array di 64 *puntatori* a `Room`, che all'avvio del server saranno tutti `null` ad eccezione della posizione `0`. In questa cella è contenuta una stanza speciale, anche detta iniziale. Si tratta di una stanza in cui non è possibile chattare e dove "si trovano" tutti i `Client` degli utenti connessi al server ma non ancora entrati in alcuna stanza.
In questo file si trovano tutte le *sezioni critiche* poiché è responsabile della gestione dei dati. Perciò si è scelto di inizializzare ed utilizzare i *semafori* (`pthread_mutex_t`) al fine di evitare situazioni *race condition* dovute al multithreading. In particolare, vi è

- Un *semaforo* `rooms_mutex` per la manipolazione dell'array di stanze (per esempio viene utilizzato quando si creano o distruggono stanze)
- Un *array* di *semafori* `room_mutexes[64]` per la manipolazione delle singole stanze (per esempio vengono utilizzati `room_mutexes[0]` e `room_mutexes[1]` per il passaggio di un client dalla stanza iniziale alla stanza con id 1)

user.h

```c
typedef struct{
    char name[32];
    char password[256];
} User;

//Construtors and Destroy
User* user_create_default();
User* user_create(const char* name, const char* password);
void user_destroy(User* self);

//Print and Debug
char* user_to_string(User* user); //Debug function
```

client.h

```c
typedef struct{
    User* user;
    int socket_id;
    unsigned int room_id;
} Client;

//Constructors and Destroy
Client* client_create(User* user, int socket_id, unsigned int room_id);
void client_destroy(Client* c);

//Print and Debug
char* client_to_string_full(Client* c); //Debug function
char* client_to_string(Client* c); //Debug function
```

room.h

```c
typedef struct {
    unsigned int id;
    char name[32];
    unsigned int clients_counter;
    Client* master_client;
    Client** clients; //Clients connected to the room
} Room;

//Constructor and Drestory
Room* room_create(unsigned int id, const char* name, Client* master_client);
void room_delete(Room* r);
void room_destroy(Room* r);

//Get
Client* room_get_client_by_id(Room* r, int client_socket_id);

//Logic
bool room_add_client(Room* r, Client* client);
bool room_remove_client(Room* r, int socket_id);
bool room_change_master(Room* r);
bool room_is_empty(Room* r);
void room_clear(Room* r);
```

La stanza iniziale ha una capienza di 256 `Client`, mentre le altre di 32 `Client`.
Le stanze sono state pensate per non richiedere controlli dall'esterno per la loro gestione:

- se il `master_client` è uscito, ne selezionerà casualmente uno nuovo
- se la stanza è vuota, si distruggerà

Logging

Per facilitare lo sviluppo, la risoluzione di problemi e il monitoraggio del sistema, si è deciso di gestire il logging attraverso un'apposita libreria esterna open source, la quale offre diversi livelli di logging, quali INFO, WARN, ERROR, DEBUG.
Questo strumento è stato cruciale nelle fasi di test e debug del sistema.
Mostriamo un esempio di Log:

Importante notare che i log di livello DEBUG vengono mostrati esclusivamente se si esegue il server con un'apposita flag.

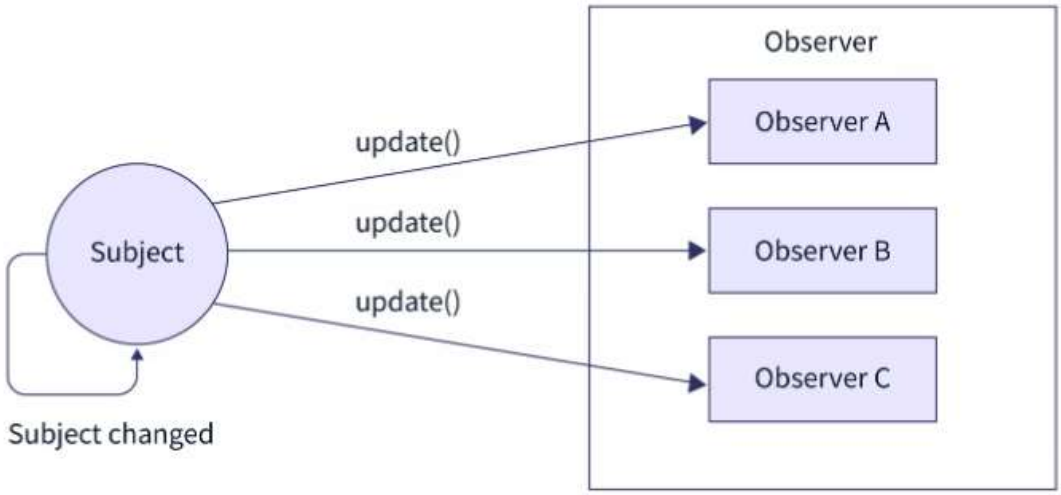
Database

Il sistema prevede anche un database in SQLite, un RDBMS che permette di usare uno specifico file come se fosse un tipico DBMS relazionale, con il beneficio di essere più leggero ed ottimizzato per query molto semplici e gestione di quantità di dati non particolarmente elevate.
Il database del sistema presenta un'unica tabella con tre colonne, utilizzata per avere persistenza delle credenziali di login e gestire lo status (online oppure offline) degli utenti.
Le password sono salvate al suo interno tramite l'algoritmo di Hashing CRC32.

Table USERS
    USERNAME: VARCHAR;
    PASSWORD: VARCHAR;
    ONLINE_STATUS: BOOL;

Client

Consiste in un'applicazione Android scritta in Java.
Si è utilizzato l'Android **SDK** fornito da Google e la nuova interfaccia grafica **Material Design 3**.
La comunicazione con il Server avviene attraverso le Java `Socket` ed il Pattern **Observer-Observable**, che risulta essere un approccio efficace per lo scambio di informazioni con un server.
Le operazioni di `Socket`, principalmente `read` e `write`, sono eseguite all'interno di appositi `Thread`, affinché non vi sia un blocco del MainThread di Android. Il sistema operativo, infatti, non tollera attese di questo tipo per motivi di sicurezza ed efficienza, ed uccide tutti i processi che tentano di farlo.



Il Pattern **Observer-Observable** è stato utilizzato per evitare problemi di sincronia tra Client e Server. Nello specifico, l'applicazione utilizza un apposito Thread all'interno del quale, tramite la funzione `readLine` (bloccante), si è in perenne attesa della Response del Server. Una volta ottenuta, vengono notificati tutti gli `Observer` in ascolto dell'effettiva ricezione di dati.
Dall'altra parte è stato scelto di avere sempre uno e un solo `Observer` in ascolto per evitare conflitti; quindi, ogni osservatore si mette in ascolto e una volta ricevuta l'informazione necessaria si rimuove dalla lista di attesa.
Tutto questo è stato possibile tramite i `MutableLiveData` della libreria **"androidx.lifecycle.MutableLiveData"**.

README

Server

Istruzioni per compilare

È necessaria la libreria make

1. Aprire la cartella Server-C
2. Usare il comando **`make`**

Istruzioni per eseguire e opzioni

Eseguire il programma tramite il comando

`bin/server`

Flag di opzioni:

`-h` per mostrare il menu d'aiuto

`-d` per mostrare i log di debug a schermo

`-i` per cambiare indirizzo IP della socket (default: localhost)

`-p` per cambiare la porta della socket (default: 9192)

```
make
bin/server -d -i "192.168.1.6" -p 9192
```

Client

Istruzioni per compilare

È necessario Android Studio con minimo SDK 26.

Di default il Client proverà a connettersi al Server tramite localhost e la porta 9192, per cambiare i valori bisogna modificare il file **Server.java** presente nella cartella "Model".

```
54 socket.connect(new InetSocketAddress("Indirizzo IP", xxxx), 15000);
```

Sostituire il campo "Indirizzo IP" con uno valido (accetta sia dominio che indirizzo ip), e il campo "xxxx" con una porta libera.

Risorse e Testi consultati

- Object-Oriented Programming with ANSI-C di Axel-Tobias Schreiner
- rxi/log.c : A simple logging library implemented in C99 su GitHub
- Linux man pages
- Introduction to Linux: A Hands on Guide di Machtelt Garrels
- Advanced Programming in the UNIX Environment, 3rd Edition di W. Richard Stevens e Stephen A. Rago