Studente: Enea Passardi, **Matricola**: 672567, **Progetto reti informatiche anno 2025-2026**

1 Overview del progetto

L'applicazione è stata sviluppata con un approccio ibrido, che combina socket TCP con socket UDP. L'idea alla base del progetto è che le comunicazioni con il server avvengano mediante l'utilizzo di socket TCP, garantendo in questo modo il trasferimento affidabile dei dati verso il server, mentre la comunicazione tra client in fase di review avvenga mediante l'utilizzo di socket UDP. I motivi di queste scelte sono molteplici:

- Il motivo per cui è stato adottato l'utilizzo di socket TCP tra client e server è dato dalla natura stessa delle interazioni, infatti un client che comunica con il server dovrà richiede e/o inviare dei dati importanti (come, ad esempio, il testo di una card), ed è quindi necessario che questo trasferimento avvenga in modo affidabile.
- Il motivo per cui, invece è stato adottato l'utilizzo di socket UDP per l'interazione con tra client in fase di review deriva dalla necessità di mantenere queste operazioni quanto più leggere possibili, senza andare ad inizializzare una connessione TCP con ogni altro client a cui si chiede la review di una card.

Dal punto di vista dell'interazione con il server, ogni client mantiene due connessioni TCP con il server, una che viene utilizzata per l'invio di comandi dal client verso il server, mentre l'altra che viene utilizzata per l'invio di comunicazioni asincrone con il server e la risposta a queste ultime. La scelta di separare i due flussi deriva proprio dalla volontà di mantenere il codice quanto più semplice possibile, separando le responsabilità su due flussi dati diversi.

2 Realizzazione del Client

La logica del client è strutturata adottando un approccio ibrido, che prevede una netta separazione tra la gestione del flusso dei comandi inviati al server e la gestione degli eventi asincroni che possono verificarsi durante l'utilizzo del sistema. In particolare:

- Il **thread** `main` gestisce l'interazione dell'utente con il server e con gli altri client; in altri termini, consente all'utente di inviare comandi al server tramite la linea di comando.
- Il **thread** `client_listener` è dedicato alla gestione degli eventi asincroni, come ad esempio l'assegnazione di una card da parte del server o la ricezione di una richiesta di review da parte di un altro client del sistema.

La scelta di utilizzare due thread distinti consente una migliore organizzazione del codice, facilitando la gestione del sistema e migliorandone la scalabilità e la modularità. Tuttavia, l'adozione di un approccio multithreading introduce una criticità legata alla gestione dell'output sul terminale: può infatti accadere che il thread asincrono stampi una notifica mentre l'utente sta digitando un comando, compromettendo la resa grafica del terminale, sebbene senza influire sulle funzionalità del sistema.

Durante la realizzazione del progetto sono state valutate due possibili soluzioni a questo problema:

- **Coda di notifiche**: ogni volta che il thread asincrono riceve una notifica, questa viene inserita in una coda. Il client, prima di acquisire l'input dell'utente, controlla la presenza di notifiche pendenti e le stampa eventualmente a schermo.
- **Implementazione di una GUI**: l'utilizzo di librerie come *ncurses* permette di realizzare una GUI minimale, suddividendo il terminale in aree dedicate.

Entrambe le soluzioni sono state tuttavia scartate in quanto poco efficienti dal punto di vista realizzativo. In particolare, *ncurses* richiede componenti aggiuntivi non sempre disponibili, mentre l'utilizzo di una coda di notifiche avrebbe potuto causare ritardi significativi nella visualizzazione dei messaggi nel caso di un client poco interattivo o inattivo.

Inoltre, la scelta di implementare l'I/O Multiplexing all'interno del thread listener consente a un singolo thread di monitorare simultaneamente più sorgenti di traffico: da un lato il canale TCP dedicato alla comunicazione con il server (ad esempio per il meccanismo di presenza tramite PING/PONG o per l'assegnazione asincrona di nuovi task), dall'altro il socket UDP utilizzato per le interazioni Peer-to-Peer durante la fase di review delle card. L'utilizzo della funzione `select()` risulta quindi giustificato dalla necessità di gestire questa eterogeneità di comunicazione senza ricorrere a un numero elevato di thread, che comporterebbe un inutile spreco di risorse di sistema.

Infine, l'impiego del protocollo UDP per la fase di review è motivato dalla natura stessa del processo: la revisione è infatti un'attività distribuita che non richiede la persistenza del server. L'uso di datagrammi consente uno scambio di messaggi rapido e leggero tra i client; inoltre, l'eventuale perdita di alcuni pacchetti non rappresenta un problema, poiché allo scadere del timeout la procedura di review può essere semplicemente riavviata.

3 Realizzazione del server

L'architettura del server è stata progettata adottando un modello *multithreaded*, con l'obiettivo di garantire una gestione efficiente della concorrenza e una buona modularità del sistema. Il thread principale svolge un ruolo puramente di coordinamento: si occupa dell'accettazione delle nuove connessioni tramite la chiamata di sistema `accept()` e delega immediatamente la gestione di ciascun client a un thread dedicato di tipo `manage_request`. Questa scelta consente alla Lavagna di rimanere costantemente reattiva, evitando che operazioni di input/output potenzialmente bloccanti o il comportamento anomalo di un singolo client possano influire sul funzionamento del sistema. Ogni sessione viene infatti gestita in modo indipendente, riducendo al minimo le interferenze tra le diverse connessioni.

Accanto ai thread dedicati alla gestione delle richieste degli utenti, il server prevede un thread separato, denominato `card_handler`, incaricato della gestione delle logiche asincrone del sistema. Questo thread opera in background ed è responsabile del monitoraggio continuo dello stato delle card e degli utenti connessi. In particolare, le sue funzionalità principali includono:

- l'assegnazione automatica delle card in stato *To Do* agli utenti disponibili, secondo l'ordine di porta stabilito;
- la gestione delle proposte di assegnamento tramite il comando asincrono `HANDLE_CARD`;
- il controllo dello stato di attività degli utenti attraverso un meccanismo di timing.

Il vantaggio di utilizzare un architettura modulare basata sull'utilizzo di diversi thread è, innanzitutto, l'isolamento delle operazioni bloccanti. Infatti operazioni come `accept`, `recv` e la risposta durante le fasi di assegnamento, sono bloccanti e, per tale motivo, è fondamentale che siano poste in modo tale da non bloccare l'intero sistema durante l'esecuzione delle operazioni di base. Inoltre, l'utilizzo di un thread dedicato per ogni client semplifica enormemente la struttura complessiva del server, evitando di dover scrivere blocchi di codice monolitici e/o difficilmente mantenibili. Infine, l'utilizzo di una struttura modulare permette anche di isolare le responsabilità, migliorando anche la scalabilità del codice.

La gestione del timer per l'invio dei ping è stata progettata con l'obiettivo di garantire *liveness* del sistema senza introdurre meccanismi complessi o costosi in termini di risorse. Tutta la gestione dei timeout e dei ping avviene all'interno del thread `card_handler` e opera ciclicamente con una frequenza prefissata definita all'interno del file `generic.h` come `HANDLER_SLEEP_TIME`. Al termine di ogni ciclo, il thread:

- Acquisce il tempo attuale.
- Confronta il valore con i timestamp associati alle card e agli utenti.
- Decide se inviare un messaggio di `PING` o se dichiarare un `TIMEOUT`.

Un approccio centralizzato consente di evitare la gestione di numerosi timer indipendenti, riducendo l'overhead complessivo del sistema.

4 Dettagli aggiuntivi

L'accesso alla lavagna è fatto mediante l'utilizzo di diverse primitive fornite all'interno del file `lavagna.h`; queste primitive si dividono in due tipologie: le primitive **thread-safe** iniziano con la parola *lavagna* (es. `lavagna_card_add`), mentre le primitive **non thread-safe** iniziano con la parola *lavagna* preceduta da un simbolo _ (es. `_remove_card`).

I messaggi che possono comparire sul terminale, sia lato client che lato server, sono di tre diverse tipologie: messaggi `[SUCCESS]` indicano che un operazione è avvenuta correttamente, i messaggi di tipo `[LOG]` sono messaggi che identificano delle notifiche di sistema e, infine, i messaggi di tipo `[ERROR]` indicano degli errori relativi a connessioni o creazione delle card.