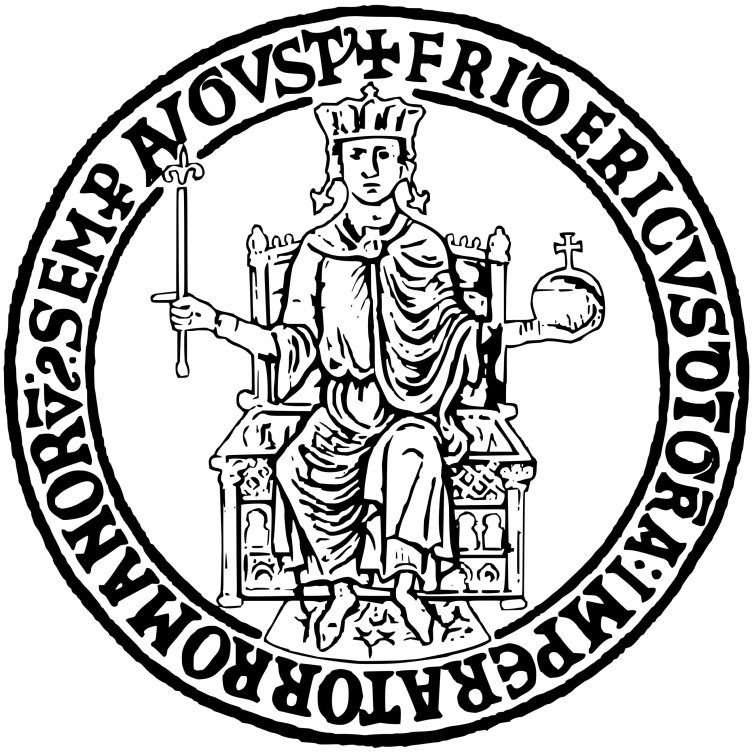

Università degli studi di Napoli Federico II

Simulazione di un server multi-client per il gioco del tris

Spera Noemi N86003717

Pollio Michela N86003697

Dipartimeno di Ingegneria Elettrica e delle Tecnologie dell'Informazione

1 Marzo 2025

Indice

1 Introduzione **3**

2 Progettazione del sistema **4**

2.1 Requisiti 4

2.2 Scelte implementative e architetturali 5

3 Dettagli di implementazione **6**

3.1 Server 6

3.1.1 Funzionalità del server 7

3.2 Client 8

3.2.1 Funzionalità del client 8

Capitolo 1

Introduzione

Il presente documento illustra le scelte implementative e architetturali adottate per lo sviluppo di un sistema multi-client, realizzato come parte del corso "Laboratorio di Sistemi Operativi" tenuto dalla prof. Rossi Alessandra nell'anno accademico 2024/2025.
L'obiettivo del sistema è permettere a più utenti di connettersi contemporaneamente ad un server centrale per avviare e gestire partite di tris.
Per garantire efficienza e scalabilità, il progetto adotta soluzioni per la gestione della concorrenza e della comunicazione tra le parti.
Nel corso del documento verranno approfonditi tutti gli aspetti che ne hanno guidato lo sviluppo.

Capitolo 2

Progettazione del sistema

2.1 Requisiti

Il progetto prevede la realizzazione di un'architettura client-server che simuli partite di tris.
All' inizio della simulazione, un numero G di giocatori si connette al server.
Ogni giocatore può decidere se creare una nuova partita, unirsi ad una partita (tramite assegnazione casuale o tramite inserimento dell'id), oppure disconnettersi.
Il creatore di una partita può accettare o rifiutare la richiesta di partecipazione alla partita da un nuovo giocatore.
Ogni qualvolta una partita viene creata, tutti i giocatori non coinvolti in un match ricevono un invito di partecipazione
Al termine di ogni partita, ciascun giocatore verrà notificato della conclusione dell'incontro e sarà automaticamente reindirizzato al menu principale dal quale potrà effettuare una nuova scelta.
I requisiti principali includono:

- **Gestione simultanea di più connessioni client**: ogni client connesso al server,infatti, viene gestito tramite un thread dedicato. Questo approccio consente al server di accettare nuove connessioni e di rispondere alle richieste di diversi client in parallelo, garantendo indipendenza e concorrenza tra le varie sessioni di gioco.

- **Sincronizzazione corretta del turno di gioco tra due giocatori**: ogni partita è gestita da un thread apposito, responsabile del coordinamento del flusso di gioco tra i due giocatori partecipanti. Le mosse vengono effettuate in modo strettamente alternato. E solo dopo aver validato e processato la mossa ricevuta, il controllo viene passato all'altro giocatore, evitando così conflitti o sovrapposizioni nei turni.

- **Notifica degli eventi in tempo reale**: è prevista la possibilità per i client di ricevere notifiche asincrone da parte del server.

Ognuno dei seguenti punti, sarà approfondito in seguito.

2.2 Scelte implementative e architetturali

Il sistema è progettato utilizzando,in particolare, un'architettura **server-multithreaded**, in cui il server gestisce le richieste provenienti dai client e coordina le sessioni di gioco. Il tutto avviene in parallelo.
Sia il client che il server sono stati sviluppati utilizzando il linguaggio di programmazione C, con il formato dei messaggi definito in modalità testuale.
La comunicazione tra il client e il server avviene attraverso **socket di rete** di tipo TCP/IP, che consentono una comunicazione veloce ed efficiente tra i processi che si trovano sulla stessa macchina.
Il modello di comunicazione tra il server e i client è principalmente **sincrono**, con il client che invia una richiesta e attende una risposta. Tuttavia, come specificato precedentemente nei requisiti, per garantire che il client non perda messaggi importanti, è stato implementato un **thread di ascolto asincrono e passivo**. Questo thread, eseguito in background, si limita a controllare la presenza di notifiche o messaggi asincroni, senza interferire con il flusso principale. In questo modo, il client può ricevere e gestire aggiornamenti in tempo reale anche durante l'attesa di input o in fasi bloccanti.
L'accesso esclusivo ad una partita sarà garantito dall'utilizzo di **mutex** definiti nella struttura della partita, che assicureranno la sincronizzazione tra i thread, evitando race conditions e conflitti.
Sono presenti anche due liste globali e i rispettivi mutex. Tali liste contengono una tutte le partite e l'altra tutti i giocatori connessi al server

Capitolo 3

Dettagli di implementazione

3.1 Server

Il server è colui che agisce come autorità centrale per il gioco.
Si occupa, infatti, di gestire le connessioni con i client, le interazioni tra client e server, ma anche le comunicazioni tra i client stessi.
Si assicura di tenere traccia dello stato del gioco e di aggiornarlo correttamente ogni qualvolta un client esegue una mossa, notificando i giocatori con i nuovi aggiornamenti.
Si preoccupa della validazione delle mosse, affinchè le regole del gioco vengano rispettate e gli è affidata la responsabilità del risultato finale. Ad ogni mossa, infatti, viene controllato(tramite funzioni apposite) se è stato proclamato un vincitore e quindi un perdente, oppure se si tratta di un pareggio. Per garantire la sincronizzazione, il server, crea e gestisce dei thread. In particolare:

- **Thread per la gestione dei client**: Quest'ultimo, come detto in precedenza, gestisce in parallelo le scelte dei client. Se un client sceglie di creare una partita viene messo in attesa utilizzando pthread_cond_wait. E solo quando la richiesta di un client viene accettata il thread del creatore viene svegliato usando pthread_cond_signal

- **Thread per la gesione delle partite**: Tale soluzione garantisce il coordinamento delle varie sessioni di gioco. Ogni partita è avviata, infatti, come un thread separato che non intacca le altre partite in corso di svolgimento. Quando una partita inizia, entrambi i thread dei client che sono in partita restano in attesa che la partita finisca. Solo quando quest'ultima termina, usando il comando pthread_cond_broadcast, andiamo a svegliare entrambi i client in attesa cosi che possano proseguire normalmente la loro esecuzione.

3.1.1 Funzionalità del server

```c
typedef struct Partita
{
    int id;
    Giocatori *giocatore[MAX_GIOCATORI];
    char griglia[N];
    StatoPartita stato;
    int turno;
    EsitoPartita risultato;
    pthread_cond_t cond;
    pthread_mutex_t mutex;
    struct Partita *next;
} Partita;
```

Listing 3.1: Struttura della partita

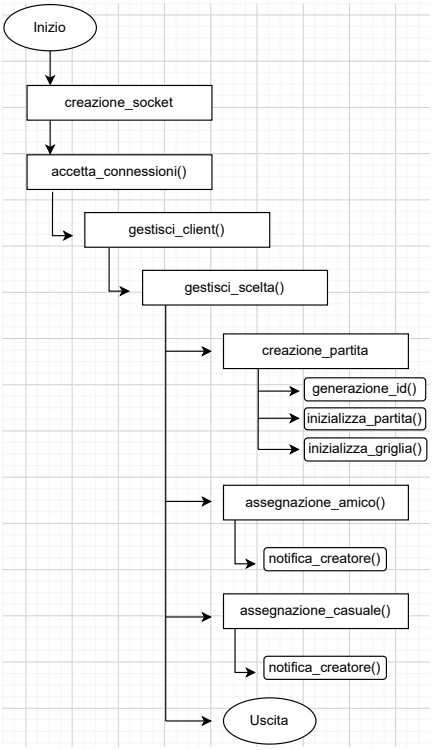

Figura 3.1: Diagramma del server

3.2 Client

Il codice client non necessita di particolari spiegazioni, ma una particolarità va evidenziata e riguarda il flusso dei messaggi. Come già detto è un flusso di tipo sincrono. Il server invia, il client riceve. Si sa esattamente il punto in cui un determinato messaggio deve essere ricevuto. Tuttavia, per gestire l'arrivo di notifiche "imprevedibili" nel codice viene creato un thread di ascolto passivo:

- **Thread per l'ascolto passivo**: Tale thread è sempre in ascolto di messaggi che potrebbero arrivare dal server, ma non li processa direttamente, almeno non tutti. Si occupa di gestire solo particolari tipi di messaggi. Il resto dei messaggi (quelli non attesi dal thread di ascolto) vengono rimandati al flusso principale del main e normalmente gestiti. Per questo lo chiamiamo thread passivo, in quanto è sempre in ascolto, ma processa solo ciò che gli spetta. Questa scelta è motivata dalla necessità di ricevere messaggi anche mentre il processo principale del client è bloccato in attesa dell'input dell'utente o impegnato in altre operazioni. Senza un thread separato dedicato all'ascolto, il client non sarebbe in grado di gestire tempestivamente messaggi non richiesti, rischiando di perderli o visualizzarli con ritardo e causandone la ricezione in punti non appropriati del codice, compromettendo la corretta sequenza delle operazioni. .

3.2.1 Funzionalità del client

```
void *ascolta_notifiche(void *arg) {
    int client_fd = *((int *)arg);
    char buffer[MAX];

    while (1) {
        int n = recv(client_fd, buffer, sizeof(buffer) - 1, MSG_PEEK);
        if (n > 0) {
            buffer[n] = '\0';
            if (strncmp(buffer, "[NOTIFICA]", 10) == 0) {
                n = recv(client_fd, buffer, sizeof(buffer) - 1, 0);
                buffer[n] = '\0';
                printf(MAGENTA "\n%s\n" RESET, buffer);
            } else {
                usleep(100 * 1000);
            }
        }
    }
    return NULL;
```

```
19 }
```

Listing 3.2: Thread di ascolto notifiche