UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

**SCUOLA POLITECNICA DELLE SCIENZE DI BASE DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL’INFORMAZIONE**

Immagine che contiene cerchio, emblema, simbolo, arte

Il contenuto generato dall'IA potrebbe non essere corretto.

***Documentazione del Software***

***Anno Accademico 2024/2025***

La partita di tris

STUDENTI

Fabrizio Bove N86004393 • Vincenzo Capasso N86004259 • Andrea Di Donato N86004934

SOMMARIO

[1 Introduzione 3](#_Toc198909772)

[2 Obiettivi del Progetto 3](#_Toc198909773)

[3 Funzionamento 4](#_Toc198909774)

[4 Architettura del sistema 5](#_Toc198909775)

[4.1 Composizione 6](#_Toc198909776)

[4.2 Thread e Concorrenza 6](#_Toc198909777)

[4.3 Errori 7](#_Toc198909778)

[4.4 Persistenza e Stato 7](#_Toc198909779)

[5 Server 8](#_Toc198909780)

[6 Client 8](#_Toc198909781)

[7 Docker 9](#_Toc198909782)

[7.1 Dockerfile 9](#_Toc198909783)

[7.2 Docker Compose 9](#_Toc198909784)

# 1 Introduzione

Il seguente progetto “La partita di Tris” consiste nello sviluppo di un server multi-client in grado di gestire partite di Tris tra due giocatori alla volta. Ogni giocatore può registrarsi al sistema, creare nuove partite o unirsi a quelle esistenti. L’architettura prevede che un giocatore possa partecipare a una sola partita per volta, ma sia libero di crearne più di una in momenti differenti.

Il server è responsabile della gestione dello stato delle partite, che può essere: *nuova creazione*, *in attesa*, *in corso* o *terminata*. A seconda dello stato di una partita, il server invia messaggi specifici ai client per notificare aggiornamenti e invitare altri utenti a partecipare. Ogni partita è identificata in modo univoco e consente una comunicazione sincrona tra i due partecipanti.

Alla fine di ogni partita, l’esito può essere: *vittoria*, *sconfitta* o *pareggio*. In base a questo, viene anche gestita l’eventuale richiesta di rivincita, con logiche differenti a seconda che vi sia un vincitore o un pareggio. Se il vincitore non è il proprietario iniziale, egli ne acquisisce il ruolo e attende un nuovo sfidante.

# 2 Obiettivi del Progetto

L’obiettivo principale di questo progetto è lo sviluppo di un’applicazione client-server multiutente che permetta di giocare a tris tra due giocatori contemporaneamente connessi al server. Il progetto prevede la realizzazione di un *server centralizzato* in grado di gestire simultaneamente più client, ciascuno dei quali può creare una partita o unirsi ad una esistente.

Gli obiettivi specifici includono:

* **Gestione delle connessioni concorrenti**: Il server deve essere in grado di gestire più client in modo parallelo, utilizzando thread separati e meccanismi di sincronizzazione (mutex e condition variable) per evitare condizioni di race.
* **Creazione e gestione delle partite**: Ogni giocatore può creare una o più partite, ma può partecipare attivamente a una sola per volta. Il creatore della partita può accettare o rifiutare richieste di partecipazione da parte di altri giocatori.
* **Gestione degli stati di gioco**: Ogni partita può trovarsi in diversi stati (creata, in attesa, in corso, terminata), così come ogni giocatore può trovarsi in stati diversi (in lobby, in partita, in attesa, vincitore). Questi stati determinano i messaggi inviati dal server ai client.
* **Determinazione dell’esito delle partite**: Al termine della partita, il server comunica l’esito (vittoria, sconfitta, pareggio) a ciascun partecipante in modo coerente e indipendente.
* **Rivincita e gestione avanzata**: (Funzionalità opzionale implementata) Il giocatore vincitore può decidere se continuare con una nuova partita, assumendo eventualmente il ruolo di proprietario. In caso di pareggio, entrambi i giocatori possono scegliere se giocare nuovamente.

# 

# 3 Funzionamento

Il gioco si sviluppa all’interno di un ecosistema client-server che permette a più utenti di connettersi contemporaneamente, creare partite o unirsi a quelle già esistenti. Una volta che il client stabilisce una connessione col server, viene richiesto all’utente di registrarsi scegliendo un nome. Questo nome sarà univoco e sarà poi utilizzato per identificare il giocatore durante l’intera sessione.

Terminata la fase di registrazione, l’utente può decidere se creare una nuova partita o consultare l’elenco di quelle già disponibili. Le partite vengono visualizzate in tempo reale grazie alla gestione dinamica del server, che mantiene aggiornato lo stato di ciascuna partita, specificando se è appena stata creata, se è in attesa di un avversario o se è in corso. Quando un secondo utente decide di unirsi a una partita esistente, viene inviata una richiesta al proprietario della partita che può accettarla o rifiutarla.

Nel momento in cui la richiesta viene accettata, la partita ha ufficialmente inizio. I due client coinvolti comunicano con il server, che agisce da mediatore tra i due, smistando le mosse e gestendo lo stato del gioco. Ogni giocatore visualizza una griglia 3x3 e inserisce la propria mossa digitando un numero da 1 a 9, che rappresenta la posizione desiderata sulla griglia. Il server riceve la mossa, la valida, la invia all’avversario e aggiorna lo stato interno della partita.

Il turno viene rispettato in modo rigoroso grazie a un controllo coordinato dal server, che assicura che solo un giocatore alla volta possa agire. Dopo ogni mossa, il server riceve l’esito della partita (vittoria, sconfitta, pareggio o continuazione), lo memorizza e lo comunica all’altro giocatore. In caso di disconnessione improvvisa di un client, il server si occupa di inviare un messaggio all’altro giocatore per informarlo e assegnargli la vittoria a tavolino.

Alla fine di ogni partita, in caso di pareggio, viene chiesto a entrambi i giocatori se desiderano una rivincita. La partita potrà riprendere solo se entrambi accettano, altrimenti il server considererà la sessione conclusa e riporterà entrambi i giocatori al menù iniziale, pronti per una nuova sfida.

# 

# 4 Architettura del sistema

Il sistema è strutturato secondo un’architettura distribuita client-server, realizzata interamente in linguaggio C e progettata per supportare più connessioni simultanee tramite l’uso del multithreading. Il server ha il compito di mantenere lo stato globale dell’applicazione, gestendo le partite attive e le informazioni relative ai giocatori connessi. Ogni client si collega al server tramite socket TCP utilizzando l’indirizzo 127.0.0.1 (localhost), e può partecipare a partite esistenti o crearne di nuove.

Il processo server è avviato su una socket in ascolto sulla porta 8080. Quando un nuovo client si connette, viene generato un thread dedicato per la gestione dell’intera sessione. Questi thread, creati in modalità detached, permettono al server di gestire in parallelo più utenti senza blocchi o interferenze, grazie a un sistema di sincronizzazione basato su mutex e condition variable.

Il client è progettato per alternare due modalità principali: la modalità di lobby e la modalità di gioco. In lobby, un thread scrittore consente all’utente di interagire con il sistema (inserendo comandi, creando partite, ecc.), mentre il thread principale rimane in ascolto passivo. Durante la partita, il thread scrittore viene terminato e tutte le interazioni sono gestite dal thread principale in modo sequenziale, per garantire coerenza nel flusso della partita.

La comunicazione tra client e server si basa sull’invio e ricezione di messaggi testuali e flag simbolici che rappresentano lo stato della partita o dell’utente. L’architettura è stata pensata per mantenere il server il più leggero possibile: non gestisce direttamente la logica del gioco, ma si limita a inoltrare le mosse, a registrare l’esito delle partite e ad aggiornare lo stato globale. L’interfaccia utente del client è testuale e minimale, ma consente una navigazione fluida tra le funzionalità offerte.

L’intero sistema è stato infine containerizzato tramite Docker, con supporto per l’avvio automatico di più client tramite docker-compose. Questo approccio ha l’obiettivo di garantire uniformità e portabilità tra diversi sistemi operativi (Windows, Linux, MacOS).

## 

## 4.1 Composizione

Il sistema è suddiviso in due componenti principali:

- ***Serve*r**: gestisce le connessioni dei client, il ciclo di vita delle partite e la sincronizzazione tra giocatori. Accetta nuovi client, gestisce le richieste di creazione e partecipazione alle partite, controlla lo stato dei giocatori e delle partite, e invia i messaggi appropriati in base agli eventi.

- **Client**: consente all’utente di interagire con il server, tramite una shell testuale in cui tramite stringhe ogni client può:

* Registrarsi con un nome utente
* Creare una partita
* Consultare le partite disponibili e i dati relativi a partite precedenti
* Inviare richieste di partecipazione a una partita
* Giocare effettivamente a Tris, inserendo le mosse
* Decidere se fare una rivincita alla fine della partita

## 4.2 Thread e Concorrenza

La sincronizzazione tra i thread è garantita mediante l’uso combinato di mutex e condition variable. Questi strumenti sono fondamentali per garantire l’accesso sicuro alle strutture dati condivise (come le liste dei giocatori e delle partite) e per coordinare il comportamento dei client durante lo svolgimento della partita, ad esempio nella gestione dei turni e nelle fasi di rivincita.

Tutti i thread, sia lato server che lato client, vengono avviati in modalità detached, poiché non è necessario recuperarne lo stato di terminazione. Ogni client connesso al server è gestito da un thread dedicato, responsabile della sua intera sessione di gioco, dalla registrazione alla disconnessione.

Per aggiornare in tempo reale tutti i client in lobby quando lo stato delle partite cambia (nuova partita, avvio, chiusura), il server utilizza il segnale SIGUSR1. I thread che modificano la lista delle partite invocano una funzione che, tramite pthread\_kill, invia SIGUSR1 a tutti i client in ascolto. Il segnale attiva l’handler send\_game, che aggiorna la lista delle partite visibile lato client.

Un meccanismo simile è usato alla registrazione di nuovi giocatori: viene inviato SIGUSR2 a tutti i thread dei client in lobby, che tramite il rispettivo handler ne notificano la presenza. Dal lato client, il thread scrittore viene terminato all’inizio della partita tramite SIGUSR1, in modo che la gestione della partita resti sequenziale nel thread principale.

Durante la partita, il thread dell’avversario viene sospeso tramite pthread\_cond\_wait su una condition variable collegata al suo stato. Viene riattivato dal thread del proprietario al termine della partita o in caso di errore, tramite pthread\_cond\_signal.

Il thread del proprietario, invece, rimane in attesa di un avversario usando una pthread\_cond\_timedwait, che permette controlli periodici sull’integrità della connessione. In caso di disconnessione improvvisa, l’error\_handler interviene sbloccando l’altro giocatore e aggiornando i dati.

Infine, per proteggere le liste condivise di giocatori e partite, vengono impiegati due mutex globali, allocati staticamente, che garantiscono accessi sicuri alle strutture dati durante le modifiche concorrenti.

## 4.3 Errori

Il sistema gestisce gli errori di comunicazione tramite una funzione error\_handler, richiamata dopo ogni send o recv fallita. Nel client, questa funzione stampa un messaggio d’errore (inattività o errore generico), chiude la socket e termina il processo. Nel server, invece, verifica se il giocatore disconnesso era in partita e, in base al ruolo, assegna la vittoria all’avversario, aggiorna gli stati e invia un flag di errore. Il thread viene poi terminato con pthread\_kill, attivando un handler che rimuove il giocatore e chiude la connessione.

Durante la partita, il server invia un flag NO\_ERROR prima di ogni mossa; in caso di disconnessione, invia ERROR, che viene gestito dal client per chiudere la partita e notificare la vittoria automatica.

Infine, il segnale SIGTERM viene gestito per terminare correttamente i processi quando Docker arresta i container.

4.4 Persistenza e Stato

Il server mantiene in memoria sia una lista dei *giocatori connessi*, tramite una lista collegata (players) che una lista delle *partite attive*, anch’essa strutturata con una lista collegata (games). Inoltre, ogni struttura, contiene informazioni rilevanti sullo stato corrente (es. socket descriptor, stato del gioco, nome giocatore, contatori di vittorie/sconfitte/pareggi).

# 

# 

# 5 Server

Nel sistema sviluppato, il server ha la responsabilità di gestire la logica di comunicazione tra client, mantenendo allo stesso tempo una struttura centralizzata per il salvataggio e l’aggiornamento delle informazioni sui giocatori e sulle partite. Ogni volta che un nuovo client si connette, il server crea un thread dedicato, che seguirà quel giocatore per l’intera durata della sessione, dalla registrazione iniziale fino alla disconnessione. L’accesso alle strutture dati condivise, come la lista dei giocatori o delle partite attive, avviene tramite mutex, per garantire la consistenza delle operazioni concorrenti.

In fase di registrazione, il thread si occupa di ricevere il nome del giocatore, verificare che non sia già in uso e allocare correttamente le risorse. Successivamente, durante la permanenza nella home, gestisce l’invio delle informazioni sulle partite disponibili e monitora costantemente lo stato dell’utente, notificandolo in tempo reale nel caso venga accettata una richiesta di partecipazione a una partita.

Una volta avviata la partita, il thread del proprietario assume il controllo della comunicazione tra i due giocatori, ricevendo e inoltrando le giocate, senza eseguire logica di verifica sulla correttezza del contenuto (che rimane a carico del client). Il thread dell’avversario resta invece in attesa passiva, pronto a ritornare attivo al termine del match. Alla fine della partita, il server aggiorna le statistiche dei giocatori coinvolti e si occupa di notificare eventuali decisioni sulla rivincita.

# 6 Client

Lato client, il processo si divide in due fasi principali: la prima comprende la registrazione e la navigazione nella home, mentre la seconda riguarda lo svolgimento effettivo della partita. All’avvio, il client crea un thread secondario che consente di ricevere aggiornamenti dal server (come notifiche di invito o cambi di stato) mentre l’utente inserisce comandi in input. Quando viene avviata una partita, il thread di scrittura viene interrotto e la gestione dell’interazione con il server passa interamente al thread principale.

Durante la partita, il client è responsabile del tracciamento locale della griglia, della verifica degli input, della determinazione dell’esito della partita e della comunicazione con il server. Se il giocatore è di turno, il client registra la mossa e invia al server la posizione e l’esito; se invece è in attesa, riceve la giocata dell’avversario e aggiorna la propria interfaccia. Al termine della partita, il client gestisce la richiesta di rivincita o il ritorno alla home, riattivando eventualmente il thread secondario per proseguire nella sessione.

# 7 Docker

Per semplificare l’esecuzione e la gestione del progetto, si è scelto di containerizzare l’intera applicazione tramite Docker. In particolare, sono stati creati due *Dockerfile*: uno per il server e uno per il client, oltre a un file *docker-compose.yml* che consente di avviare più container in modo coordinato e connessi tra loro, sfruttando la rete interna di Docker.

## 7.1 Dockerfile

I due Dockerfile, rispettivamente per il client e per il server, sono molto simili e contengono solo le istruzioni essenziali. La base utilizzata è l’immagine Ubuntu, selezionata per la sua stabilità e facilità d’uso, a discapito di qualche megabyte in più rispetto ad alternative più leggere come Arch Linux.

All’interno di ciascun Dockerfile vengono installati solo i pacchetti necessari, ossia *gcc* e *make*, tramite il comando apt-get. Una volta completata l’installazione, si procede con la rimozione della cache per ridurre il peso finale dell’immagine.

La directory di lavoro impostata è /app, nella quale vengono copiati tutti i file del progetto. A quel punto, viene eseguito make per compilare il codice e generare l’eseguibile corrispondente. Infine, l’istruzione CMD definisce il comando da avviare al boot del container: ./server per il server, ./client per il client.

## 7.2 Docker Compose

Per orchestrare l’intero sistema è stato utilizzato il file docker-compose.yml, posizionato nella directory principale del progetto. Questo file definisce due servizi distinti: server e client, specificando per ciascuno la directory di build (cioè dove si trovano i rispettivi Dockerfile).

L’avvio del client è subordinato all’esecuzione del server, grazie all’istruzione depends\_on: server, che evita problemi di connessione. Inoltre, per rendere il client interattivo da terminale, sono stati abilitati lo *standard input* (stdin\_open: true) e *un emulatore di terminale* (tty: true), entrambi necessari per permettere l’interazione diretta con l’utente attraverso docker attach.

Infine, si è scelto di usare network\_mode: "container:server", che permette al container client di condividere lo stesso spazio di rete del server. In questo modo, il client può comunicare con il server utilizzando 127.0.0.1 come indirizzo IP, evitando l’esposizione di porte sull’host. Questa modalità è stata preferita rispetto alla creazione di un network personalizzato, più indicata in scenari complessi con molti servizi, risultando invece una soluzione più compatta e adeguata ad un progetto come il nostro.