



UNIVERSITÀ DEGLI STUDI DI NAPOLI  
**FEDERICO II**

Progettazione e sviluppo di un sistema  
concorrente e distribuito (Client-Server) per la  
gestione di partite a Tris

CdL Triennale in Informatica  
CORSO DI LABORATORIO DI SISTEMI OPERATIVI  
PAOLO LIBERTI  
N86004255  
EMANUELE MILANO  
N86004210

ANNO ACCADEMICO: 2024/2025

# INDICE

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Struttura del progetto . . . . .	2
<b>2</b>	<b>Progettazione</b>	<b>3</b>
2.1	Funzionalità principali . . . . .	3
2.2	Responsabilità del Server . . . . .	4
2.3	Responsabilità del Client . . . . .	4
<b>3</b>	<b>Implementazione</b>	<b>5</b>
3.1	Comunicazione Socket . . . . .	5
3.2	Strutture Dati . . . . .	6
3.3	Logica Server (funzioni.c) . . . . .	6
<b>4</b>	<b>Docker</b>	<b>7</b>
4.1	Dockerfile . . . . .	7
4.2	Docker Compose . . . . .	7
<b>5</b>	<b>Manuale d'uso</b>	<b>8</b>
5.1	windows-run.bat . . . . .	8
5.2	run.ps1 . . . . .	8
5.3	linux-run.sh . . . . .	8

# 1 Introduzione

L'elaborato si propone di descrivere le scelte di progettazione ed implementazione di un sistema Client-Server per la gestione di partite a tris in modalità concorrente. Lo sviluppo in ogni sua fase è stato supportato dall'utilizzo di Git e della piattaforma GitHub per la condivisione dei file ed organizzazione del flusso di lavoro. Come scelte preliminari si è deciso di utilizzare il linguaggio C per l'implementazione di entrambi i lati Client e Server, l'ambiente d'esecuzione è definito tramite Docker Compose e l'avvio è automatizzato da uno dei due script file a seconda del sistema operativo host.

## 1.1 Struttura del progetto

```
LS024-25/
|-- Server/
|   |-- Dockerfile
|   |-- main.c
|   |-- funzioni.c
|   |-- funzioni.h
|   |-- strutturedati.h
|   |-- winsock_posix_compat.h
|-- Client/
|   |-- Dockerfile
|   |-- main.c
|   |-- funzioni.c
|   |-- funzioni.h
|   |-- strutturedati.h
|   |-- winsock_posix_compat.h
|-- windows-run.bat
|-- linux-run.sh
|-- run.ps1
```

```
|-- docker-compose.yml
```

## 2 Progettazione

L'architettura del sistema è stata concepita seguendo un modello Client-Server per separare in modo chiaro le responsabilità e le logiche di gioco dalla presentazione all'utente. Questa scelta non solo assicura la gestione concorrente di più partite e giocatori, ma garantisce anche una solida base per la scalabilità futura. Abbiamo concentrato l'attenzione sulla portabilità e sulla robustezza del codice, elementi cruciali per il corretto funzionamento su piattaforme diverse e in ambienti dinamici.

### 2.1 Funzionalità principali

- **Connessioni Multiple:** Il server gestisce contemporaneamente più client.
- **Matchmaking:** Sistema di coda semplice per i client che desiderano giocare.
- **Interazione Asincrona:** Il client utilizza un thread dedicato per la ricezione dei messaggi, mantenendo il thread principale libero per l'invio dei comandi utente.
- **Stati del Gioco:** Il gioco è gestito tramite un'enumerazione degli stati (GameState per il server e ClientState per il client) che permette di controllare il flusso della partita, dalla creazione alla sua conclusione.
- **Portabilità:** Il codice è stato scritto per essere compilabile ed eseguibile su Windows e Linux senza modifiche. Questo è stato possibile grazie a un layer di astrazione (winsock posix compat.h) che gestisce le differenze tra le API Winsock e quelle POSIX.

## 2.2 Responsabilità del Server

Il server agisce come il motore centrale del gioco, gestendo tutte le operazioni che richiedono un coordinamento tra i client. Le sue responsabilità principali sono:

### Gestione Connessioni:

- **Accettazione:** Si mette in ascolto sulla porta 9100 e accetta nuove connessioni in entrata dai client.
- **Gestione Sessioni:** Mantiene un elenco di tutti i client connessi nell'array globale `clients[]`.
- **Logica di Gioco:** Implementa le regole del gioco (turni, controllo vittoria/pareggio) e gestisce l'aggiornamento del tabellone.
- **Matchmaking e Stato:** Gestisce la creazione delle partite, la ricerca di avversari e la transizione degli stati di gioco.
- **Disconnessioni:** Rileva le disconnessioni dei client (quando `recv()` restituisce un valore minore o uguale a 0) e rimuove il socket corrispondente dall'elenco.

## 2.3 Responsabilità del Client

Il Client è l'interfaccia utente e ha le seguenti responsabilità:

- **Connessione:** Risolve l'hostname server (come definito in Docker Compose) e stabilisce la connessione TCP sulla porta 9100.
- **Input/Output (I/O):** Legge i comandi dall'input standard (`stdin`) e li invia al Server.
- **Ricerca Asincrona:** Utilizza un thread dedicato per ricevere e stampare i messaggi dal Server senza bloccare il thread principale

- **Visualizzazione:** Stampa i messaggi di benvenuto, i comandi disponibili e l'output del gioco (tabellone, stato).

## 3 Implementazione

### 3.1 Comunicazione Socket

Il progetto adotta un approccio di astrazione del sistema operativo per garantire la compatibilità tra Windows e sistemi POSIX (Linux/macOS).

**Winsock posix compat.h:** Questo header definisce tipi e macro condizionali (ifdef WIN32) per uniformare le API:

- **Tipo Socket:** sock-t è definito come SOCKET su Windows e int su POSIX.
- **Funzioni/Macro Thread:** Macro come CREATE RECEIVE THREAD utilizzano CreateThread su Windows e pthread create su POSIX.

**Risoluzione Server(Client):** Il client utilizza gethostbyname(SERVER IP) dove SERVER IP è "server". Questo si basa sul DNS interno di Docker Compose per la risoluzione del nome del container.

## 3.2 Strutture Dati

La gestione delle strutture dati viene effettuata in un file apposito stutturedati.h. Questo file funge da "dizionario" del progetto, contenendo le definizioni condivise tra client e server.

```
typedef enum { NUOVA, IN_ATTESA, IN_CORSO, TERMINATA } GameState;

typedef struct {
    int id[256][2];
    sock_t owner; // Sostituito con il tipo compatibile
    sock_t challenger; // Sostituito con il tipo compatibile
    sock_t pendingChallenger; // Sostituito con il tipo compatibile
    GameState state;
    char board[3][3];
    int turn;
} Game;
```

Figure 1: Struttura Dati Server

```
#define SERVER_IP "server"

// Stati del client
typedef enum { MENU, IN_PARTITA, FINE_PARTITA } ClientState;

// Variabili globali (visibili nei file .c)
```

Figure 2: Struttura Dati Client

## 3.3 Logica Server (funzioni.c)

Il Server implementa le funzioni principali di gestione del gioco:

- **createGame()/joinGame()**: Gestiscono la creazione di una nuova partita e la ricerca di una partita in stato IN ATTESA per il matchmaking.
- **handleOwnerResponse()**: Permette all'owner di accettare o rifiutare un pendingChallenger, modificando lo stato a IN CORSO in caso di accettazione.
- **playTurn()**: Verifica la validità della mossa, aggiorna il tabellone e controlla la vittoria o il pareggio.
- **checkVictory()/checkDraw()**: Funzioni deterministiche che verificano le condizioni di fine partita.

## 4 Docker

L'uso di Docker e Docker Compose containerizza l'ambiente di sviluppo e garantisce un deployment coerente su qualsiasi sistema operativo (Windows, Linux, macOS).

### 4.1 Dockerfile

Sia il Client che il Server utilizzano un Dockerfile quasi identico per la compilazione e l'esecuzione:

- **Immagine Base:** gcc:12
- **Compilazione:** RUN gcc -o `!nome!` main.c funzioni.c -Wall -O2 -pthread. Il flag -pthread è necessario per la gestione dei thread.
- **Porta (Server):** Il Server espone la porta 9100

### 4.2 Docker Compose

Il file docker-compose.yml definisce i due servizi principali e la loro interazione:

```
# docker-compose.yml
1  # docker-compose.yml
2  services:
3    server:
4      build:
5        context: ./server
6        dockerfile: Dockerfile
7        container_name: my_server
8        ports:
9          - "9100:9100"
10
11   client:
12     build:
13       context: ./client
14       dockerfile: Dockerfile
15       container_name: my_client
16       depends_on:
17         - server
18
```

Figure 3: Docker Compose

- **Networking:** Docker Compose crea una rete bridge interna dove il container client può raggiungere il container server utilizzando il suo hostname (il nome del servizio, server) sulla porta 9100.

## 5 Manuale d'uso

Gli script automatizzano le operazioni di docker-compose e l'apertura di finestre separate per i client.

### 5.1 windows-run.bat

Questo è un semplice file batch che avvia lo script PowerShell, assicurando che la policy di esecuzione sia bypassata e che il sistema si metta in pausa dopo l'esecuzione.

### 5.2 run.ps1

Gestisce l'ambiente Docker Compose e l'avvio multiplo dei client in nuove finestre PowerShell:

- Chiede all'utente il numero di client (clientCount)
- Esegue docker-compose down e docker-compose build per pulizia e aggiornamento.
- Avvia il Server (docker-compose up server) in un processo PowerShell separato (Start-Process powershell -ArgumentList "-NoExit").
- Esegue i Client in loop, ognuno in una nuova finestra PowerShell interattiva, usando docker-compose run -rm client.

### 5.3 linux-run.sh

Script per l'avvio su sistemi POSIX, utilizza docker compose in stile UNIX:

- Chiede il numero di client (NUM CLIENTS) e verifica l'input.
- Avvia il Server in modalità detached (-d)
- Avvia i Client in un loop, utilizzando gnome-terminal (o un terminale equivalente) per aprire ogni client in una sessione interattiva separata, utilizzando docker compose run -rm -service-ports client.