Università degli studi di Napoli Federico II

Corso di Laurea Triennale in Informatica

Tic-Tac-Toe

Beatrice Chiara Morgillo
N86004821
ANNO ACCADEMICO 2024/25

Indice

Capitolo 1

Introduzione e traccia

Il seguente progetto descrive la progettazione e implementazione di un gioco del tris, con un'architettura client-server multi-client, in modo tale da garantire una gestione efficiente delle connessioni e delle partite tra più giocatori. Il sistema è composto da un server centrale che gestisce la logica di gioco e le comunicazioni tra i client, i quali rappresentano i giocatori. Questa architettura permette di scalare facilmente il numero di giocatori e di gestire le interazioni in tempo reale. L'intero progetto è stato realizzato utilizzando il linguaggio di programmazione C, scelto per la sua efficienza e controllo a basso livello sulle risorse di sistema. La comunicazione tra client e server avviene tramite socket TCP, garantendo una connessione affidabile e continua durante le partite. Docker-compose è stato utilizzato per semplificare la gestione dei container e delle dipendenze del progetto, consentendo un avvio rapido e una configurazione coerente dell'ambiente di sviluppo.

1.1 Struttura del Progetto

Di seguito è riportata la struttura completa del progetto, organizzata in modo da separare chiaramente le componenti server e client:

Struttura del Progetto

```
ProgettoLSO/
 server/
    src/
       main.c              # Server main entry point
       network.c           # Network handling and client threads
       lobby.c             # Client lobby management
       game_manager.c      # Game logic and state management
       headers/
           network.h       # Network function declarations
           lobby.h         # Lobby management declarations
           game_manager.h  # Game management declarations
    .dockerignore
    Dockerfile              # Server container configuration
    Makefile                # Server build configuration
    server                  # Compiled server executable
 client/
    src/
       main.c              # Client main and UI
       network.c           # Client network communication
       game_logic.c        # Game state and validation
       ui.c                # User interface functions
       headers/
           network.h       # Network function declarations
           game_logic.h    # Game logic declarations
           ui.h            # UI function declarations
    .dockerignore
    Dockerfile              # Client container configuration
    Makefile                # Client build configuration
    client                  # Compiled client executable
 .env                        # Environment variables
 .gitignore                  # Git ignore rules
 .vscode/                    # VS Code configuration
 docker-compose.yml          # Multi-container orchestration
 Mac.sh                      # macOS launcher script
 Unix.sh                     # Linux launcher script
 run.sh                      # Universal Unix launcher script
 runWin.ps1                  # Windows PowerShell script
 Windows.bat                 # Windows batch launcher
```

Capitolo 2

Progettazione del Gioco

2.1 Requisiti Individuati

Il sistema deve implementare un gioco multiplayer del Tris (o Tic-Tac-Toe) con le seguenti caratteristiche:

- Supporto per più partite simultanee
- Gestione delle connessioni dei client
- Interfaccia utente intuitiva
- Logica di gioco corretta e gestione degli stati

Il gioco può avere *N partite contemporaneamente*, a ogni partita è associato un ID univoco e possono parteciparvi massimo 2 giocatori. La distinzione tra i 2 giocatori è netta: c'è l'host della partita, il proprietario che ha avviato la partita, e c'è il secondo giocatore, che si unisce alla partita che vede disponibile. Il sistema gestisce diversi stati per il ciclo di vita delle partite:

`WAITING` Partita creata dall'host, in attesa del secondo giocatore

`PENDING` Un giocatore ha richiesto di unirsi, in attesa di approvazione dell'host

`PLAYING` Partita in corso con entrambi i giocatori attivi

`OVER` Partita terminata (vittoria di uno dei giocatori o pareggio)

`REMATCH_REQUESTED` Richiesta di rivincita in attesa di risposta

Avviando il sistema, ogni client può scegliere di:

- Creare una nuova partita (diventando host);
- Unirsi a una partita esistente, tramite l'ID della partita (visibile all'occorrenza);
- Uscire dal sistema.

Con la creazione di una partita e la partecipazione, da parte dei due client, man mano che va avanti il gioco c'è una schermata di rivincita, visibile quando la partita finisce, altrimenti i due client vengono reindirizzati al menù principale.

2.2 Scelte Architetturali

Il sistema adotta un'architettura client-server centralizzata con gestione multi-threading per supportare connessioni simultanee.

La logica di gioco è completamente gestita dal server per garantire consistenza e prevenire cheating:

Validazione mosse Ogni mossa viene validata dal server prima dell'applicazione

Gestione turni Il server coordina i turni tra i due giocatori

Calcolo vittorie Controllo automatico delle condizioni di vittoria/pareggio

Stato sincronizzato Lo stato della griglia è mantenuto esclusivamente sul server

I client possono eseguire le seguenti operazioni durante il ciclo di vita di una sessione:

- **Registrazione**: Autenticazione con nome utente univoco
- **Visualizzazione lobby**: Lista delle partite disponibili in tempo reale
- **Creazione partita**: Diventare host di una nuova partita
- **Richiesta join**: Richiedere di unirsi a una partita esistente

Gestione Partita

- **Approvazione join**: L'host può approvare/rifiutare richieste di partecipazione
- **Esecuzione mosse**: Posizionamento del proprio simbolo (X o O) sulla griglia 3x3
- **Visualizzazione stato**: Aggiornamenti in tempo reale dello stato della partita
- **Sistema rematch**: Richiesta di rivincita al termine di una partita
- **Abbandono partita**: Possibilità di lasciare una partita in corso
- Ogni client connesso riceve un thread dedicato (`network_handle_client_thread`)
- I thread sono detached per auto-cleanup alla disconnessione
- Timeout di registrazione (30 secondi) per prevenire connessioni zombie
- Gestione graceful shutdown tramite signal handler cross-platform

2.3 Componenti Principali del Sistema

Il sistema è strutturato attorno a tre componenti fondamentali, ognuna rappresentata da specifiche strutture dati:

Struttura Client (Giocatore)

```
typedef struct {
    socket_t client_fd;
    struct sockaddr_in udp_addr;
    char name[MAX_NAME_LEN];
    int game_id;
    char symbol;
    int is_active;
    thread_t thread;
} Client;
```

La struttura `Client` rappresenta ogni giocatore connesso al server, mantenendo le informazioni di connessione, identificazione e stato nel sistema.

Struttura Game (Partita)

```
typedef struct {
    int game_id;
    Client *player1;
    Client *player2;
    Client *pending_player;
    char board[3][3];
    char current_player;
    GameState state;
    int rematch_requests;
    int rematch_declined;
    Client *rematch_requester;
    mutex_t game_mutex;
    time_t creation_time;
} Game;
```

Ogni partita mantiene il proprio stato indipendente, gestisce i due giocatori coinvolti e coordina il sistema di rematch.

Sistema Lobby (Gestione Centralizzata)

```
static Client *clients[MAX_CLIENTS];
static mutex_t lobby_mutex;

Client* lobby_add_client(socket_t client_fd, const char *name);
void lobby_remove_client(Client *client);
Client* lobby_find_client_by_name(const char *name);
void lobby_broadcast_message(const char *message);
```

La lobby gestisce centralmente tutti i client connessi, coordinando registrazione, ricerca e comunicazione tra giocatori e partite.

Capitolo 3

Implementazione e Comunicazione

3.1 Architettura di Comunicazione

La comunicazione tra client e server avviene tramite un sistema dual-protocol che utilizza sia socket TCP che UDP per ottimizzare diverse tipologie di messaggi. Il server ascolta le connessioni TCP sulla porta 8080 (configurabile tramite variabili d'ambiente) e gestisce contemporaneamente un socket UDP per comunicazioni asincrone rapide.

Per garantire un riavvio rapido del server senza errori di binding, viene abilitata l'opzione `SO_REUSEADDR` sui socket TCP. Il server implementa inoltre socket non-bloccanti per prevenire deadlock durante le operazioni di rete e utilizza timeout configurabili per la gestione delle connessioni.

3.2 Gestione Multi-Threading

Ogni nuova connessione TCP accettata dal thread principale genera la creazione di un thread dedicato tramite la funzione `network_handle_client_thread`. Questo thread gestisce l'intero ciclo di vita del client, dalla registrazione iniziale fino alla disconnessione, utilizzando una struttura `Client` per mantenere tutte le informazioni relative alla connessione.

Il sistema implementa tre tipi di thread principali:

- **Thread principale**: Gestisce l'accettazione di nuove connessioni TCP tramite `select()` non-bloccante
- **Thread per client**: Ogni client connesso riceve un thread dedicato che gestisce la comunicazione bidirezionale
- **Thread UDP**: Un thread separato (`network_handle_udp_thread`) gestisce tutte le comunicazioni UDP asincrone

I thread vengono creati in modalità detached per garantire l'auto-cleanup delle risorse alla disconnessione del client, evitando memory leak e accumulo di thread zombie.

3.3 Protocollo di Comunicazione

La comunicazione si basa su un protocollo testuale definito nel file `protocollo.h`, che standardizza tutti i comandi e le risposte del sistema. I messaggi seguono il formato `"COMMAND:parameters"` per garantire parsing uniforme e gestione degli errori.

Il sistema utilizza due canali di comunicazione:

TCP Per messaggi critici che richiedono affidabilità (mosse di gioco, creazione partite, join, autenticazione)

UDP Per aggiornamenti di stato rapidi e non critici (ping/pong, sincronizzazione board, notifiche real-time)

3.4 Gestione delle Partite

A differenza di sistemi che creano thread dedicati per ogni partita, il presente progetto mantiene la gestione delle partite all'interno dei thread client esistenti, coordinati dalla lobby centrale. Quando due giocatori si uniscono a una partita, la comunicazione tra loro viene orchestrata dal sistema lobby che:

- Coordina l'approvazione del join da parte dell'host
- Sincronizza i turni tra i due giocatori

- Gestisce la validazione e l'applicazione delle mosse
- Coordina il sistema di rematch al termine della partita

Ogni partita mantiene il proprio stato indipendente tramite mutex dedicati (`game_mutex`) per garantire accesso thread-safe alle strutture dati condivise, mentre la lobby utilizza un mutex globale (`lobby_mutex`) per proteggere l'array dei client connessi.

3.5 Sincronizzazione e Thread Safety

Il sistema implementa una strategia di sincronizzazione multi-livello:

- **Lobby mutex**: Protegge l'accesso all'array globale dei client
- **Game mutex**: Ogni partita ha un mutex dedicato per proteggere il proprio stato
- **Cross-platform abstraction**: Utilizza `CRITICAL_SECTION` su Windows e `pthread_mutex_t` su sistemi Unix

La gestione dei timeout (30 secondi per la registrazione) previene connessioni zombie, mentre il sistema di signal handling cross-platform garantisce un graceful shutdown del server su `SIGINT` e `SIGTERM`.

Capitolo 4

Docker

Il file docker-compose.yml, situato nella root del progetto, definisce due servizi: server e client, specificando per ciascuno la directory dove si trova il rispettivo Dockerfile. Il servizio client è configurato per avviarsi solo dopo il server (depends_on) e ha l'input da terminale abilitato (stdin_open, tty) per poter interagire con l'utente.

L'opzione network_mode: "container:server" consente al client di condividere lo stesso stack di rete del server, permettendo la comunicazione diretta tra i due container tramite 127.0.0.1. Il server espone la porta 8080 sulla macchina host per consentire connessioni da client esterni al sistema Docker.