Progetto Laboratorio di Sistemi Operativi

Progetto 1: La partita di Tris

Anno accademico 2024/2025

Corso informatica

Autori:
Mirko Prevenzano N86004082
Mattia Di Prisco N86004267

Contents

1 Introduzione

1.1 Analisi e specifica dei requisiti

- Si richiede la realizzazione di un server multi-client per giocare a Tris
- Un giocatore per accedere all'applicazione deve inserire un nickname
- Un giocatore può creare una o più partite, ma può giocare solo una partita per volta
- Un giocatore può richiedere di partecipare ad una o più partite, se una di queste richieste è accettata dal giocatore proprietario le altre richieste verranno cancellate.
- Il proprietario di una partita può ricevere una o più richieste per una partita, al momento che una richiesta è accettata le restanti verranno rifiutate automaticamente. Il proprietario può anche rifiutare una richiesta ricevuta.
- Gli stati di un gioco possono essere i seguenti: terminata, in corso, in attesa e nuova creazione
- Gli stati di terminazione di una partita possono essere i seguenti rispetto al giocatore: vittoria, sconfitta e pareggio
- I giocatori di ogni partita, al termine di quest'ultima possono scegliere se iniziare o meno un'altra partita. Il vincitore può decidere se fare un'altra partita e in questo caso diventerà il proprietario della partita. Il perdente lascia la partita. Se la partita finisce in pareggio i giocatori possono decidere se effettuare la rivincita, solo nel caso in cui entrambi i giocatori accettano. Se uno dei due rifiuta, l'altro client sarà notificato e tornano nella home_page
- Si possono aver al più 8 client connessi contemporanemante, i client in totale possono creare al più 20 partite ed ogni partita può avere 7 richieste di unione.

1.2 Architettura generale

Il sistema adotta un'architettura client-server. Il server è multi-client quindi gestisce più client simultaneamente. La comunicazione tra client e server avviene attraverso connessione TCP, garantendo connessione affidabile attraverso socket. Abbiamo scelto di optare, per lo scambio di messaggi, l'uso del formato JSON per avere una struttura ben definita.

1.3 Tecnologia utilizzate

1.3.1 Lato Server

- **Linguaggio di programmazione**: C
- **Threading**: POSIX thread (pthread)
- **Gestione JSON**: libreria cJSON con funzioni per parsing e generazione JSON
- **Sincronizzazione per i dati**: uso di mutex POSIX (permette di proteggere l'accesso a informazioni condivise mentre sono usate da un determinato client)

1.3.2 Lato client

- **Linguaggio di programmazione**: Python 3, la scelta di usare questo linguaggio ricade per la sua grande quantità di librerie e la sua efficienza. È un linguaggio mai usato per corsi precedenti ed è nata la curiosità di provarlo.
- **Framework GUI**: Tkinter con TTK per l'interfaccia

1.4 Strutture dati

1.4.1 Giocatore

Struttura dati utilizzata per rappresentare un singolo giocatore. Contiene i seguenti campi:

- `int socket`: identificativo del socket associato al giocatore, utilizzato per la comunicazione con il server.
- `int id`: identificatore univoco del giocatore all'interno del sistema.
- `char nome[30]`: stringa contenente il nome del giocatore.
- `stato_giocatore`: tipo enumerato che può assumere i valori `IN_GIOCO` o `IN_HOME`, indicante lo stato attuale del giocatore.

1.4.2 Richiesta

Struttura dati che modella una richiesta effettuata da un giocatore. Comprende i seguenti campi:

- `GIOCATORE* giocatore`: puntatore alla struttura del giocatore che ha inoltrato la richiesta.
- `StatoRichiesta`: tipo enumerato che può assumere i valori `IN_ATTESA`, `RIFIUTATA` o `ACCETTATA`, rappresentante lo stato corrente della richiesta.

1.4.3 Partita

Struttura dati che descrive una partita. Include i seguenti campi:

- `int id`: identificatore univoco della partita.
- `GIOCATORE* giocatorePartecipante[2]`: array di due puntatori a strutture `GIOCATORE`, corrispondenti ai due partecipanti. Il primo elemento identifica il proprietario della partita.
- `Esito esito`: tipo enumerato che può assumere i valori `VITTORIA`, `PAREGGIO`, `SCONFITTA`, `NESSUN_ESITO`, indicante l'esito finale della partita.
- `STATO stato_partita`: tipo enumerato che può assumere i valori `IN_CORSO` o `IN_ATTESA`, rappresentante lo stato attuale della partita.
- `RICHIESTA* richieste[MAX_RICHIESTE]`: array di puntatori a strutture `RICHIESTA`, contenente le richieste associate alla partita.
- `int numero_richieste`: numero di richieste attualmente memorizzate nell'array delle richieste.
- `int turno`: indicatore del turno del giocatore (0 per il primo giocatore, 1 per il secondo giocatore).
- `int votiPareggio`: variabile fondamentale per tenere il conto dei giocatori che vogliono effettuare la rivincita dopo che la partita sia terminata in pareggio.
- `sem_t semaforo`: essenziale per garantire che le mosse di gioco, i cambi di turno e la gestione dello stato della partita avvengano in modo sicuro e ordinato nell'ambiente multi-thread.

2 Gestione del server

L'architettura del server è stata progettata per garantire robustezza, modularità e prestazioni in un contesto multi-threaded. Il modello di threading si sviluppa su tre livelli principali. Il thread principale si occupa esclusivamente di accettare le connessioni in entrata (tramite accept()), mantenendo la gestione delle nuove connessioni semplice e isolata. Ogni nuova connessione genera un thread dedicato alla configurazione iniziale del client e alla gestione della sua struttura interna (GIOCATORE), mentre l'elaborazione operativa viene delegata a thread specializzati. In particolare, il thread "Router" gestisce i messaggi in ingresso applicando una logica di routing basata sui path JSON, garantendo reattività e isolamento tra i client.

Un ulteriore thread dedicato monitora proattivamente le disconnessioni, utilizzando select() con MSG_PEEK per tenere sotto controllo lo stato dei socket. La scelta di usare select() consente di monitorare lo stato di un socket senza alterare i dati ricevuti, lasciando intatto il flusso di comunicazione per il thread di

```c
void checkRouter(char* buffer, GIOCATORE*nuovo_giocatore, int socket_nuovo, int *leave_flag){
    cJSON *path = cJSON_GetObjectItem(json, "path");

    if (strcmp(path->valuestring, "/waiting_games") == 0) {
        handlerInviaGames(socket_nuovo);
    }
    else if(strcmp(path->valuestring, "/new_game") == 0) {
        new_game(leave_flag,buffer,nuovo_giocatore);
    }
    // ... altri endpoint
```

routing. Ciò è possibile perchè effettua controlli passivi periodici senza bloccare l'esecuzione del thread.

La sincronizzazione è gestita tramite un sistema di mutex , che proteggono le strutture condivise (giocatori e partite), riducendo al minimo il rischio di contese e massimizzando il throughput (quantità di lavoro utile completato in un dato intervallo di tempo). Questo approccio, insieme a un ordine coerente di acquisizione dei lock, aiuta a prevenire situazioni di deadlock.

Dal punto di vista della gestione delle risorse, il server adotta strategie di cleanup automatico, come l'uso di pthread_detach() per il rilascio dei thread, e gestisce in modo sicuro eventuali errori di allocazione dinamica. La comunicazione client-server si basa su un protocollo JSON standardizzato, leggibile e facilmente estendibile, che sfrutta una struttura coerente per ogni messaggio, validata tramite la libreria cJSON. Questo schema facilita la manutenzione, il debug e l'interoperabilità cross-platform.

```c
int numeroConnessioni = 0;
int numeroPartite = 0;

GIOCO* Partite[MAX_GAME] = { NULL };
GIOCATORE* Giocatori[MAX_GIOCATORI] = { NULL };

pthread_mutex_t playerListLock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t gameListLock = PTHREAD_MUTEX_INITIALIZER;
```

Figure 1: Rappresentazione dei dati condivisi globalmente, e i mutex che ne gestiscono gli accessi

2.0.1 Vantaggi multi-threading

- Più client possono interagire contemporaneamente
- Ogni client è gestito in modo indipendente evitando interferenze
- Il server è ingrando di rispondere a tutti i client
- Maggiore semplicità nell'implementazione rispetto ad un single-thread che necessita di una gestione manuale delle connessioni

2.1 Implementazione connessione

Il server è stato progettato per gestire connessioni multiple da parte di client utilizzando il protocollo TCP. A tal fine, viene adottato un approccio multi-thread,

in cui ogni client connesso viene gestito da un thread separato, permettendo così l'esecuzione concorrente delle richieste.

All'avvio, il server effettua le seguenti operazioni:

1. Crea un socket IPv4 (`AF_INET`) con protocollo TCP (`SOCK_STREAM`) tramite la system call `socket()`.
2. Collega il socket all'indirizzo locale `127.0.0.1` sulla porta `8080` utilizzando la system call `bind()`.
3. Pone il socket in modalità di ascolto attraverso la system call `listen()`, specificando come valore del backlog un massimo di `8` connessioni pendenti.
4. Entra in un ciclo di accettazione delle connessioni: ogni volta che un client richiede una connessione, il server la accetta tramite la system call `accept()`.
5. Per ogni connessione accettata, viene creato un nuovo thread tramite `pthread_create()`, incaricato di gestire la comunicazione con il client.
6. Il thread viene immediatamente staccato mediante la chiamata `pthread_detach()`, in modo tale che le risorse siano rilasciate automaticamente al termine della sua esecuzione.

L'implementazione fa uso delle seguenti system call POSIX per la gestione delle connessioni: socket(), bind(), listen(), accept(), pthread_create(), pthread_detatch

3 Gestione Client

Il client è stato sviluppato in Python, utilizzando un'interfaccia grafica realizzata con Tkinter. L'architettura segue un pattern MVC modificato, dove le pagine fungono da viste (View), la logica applicativa è centralizzata nei controller (come MainApp e HomeActions), e i dati condivisi tra le schermate rappresentano il modello (Model). Il progetto è organizzato in modo modulare: una cartella principale contiene i file core (client_gui.py, client_network.py, server_polling.py, constants.py), mentre la logica delle pagine è suddivisa in sottocartelle specifiche per ogni area funzionale (login, home, game).

La comunicazione tra client e server avviene tramite socket TCP, utilizzando la funzione select() per implementare una gestione non bloccante, che assicura un'interfaccia fluida anche in caso di ritardi di rete. I messaggi scambiati sono in formato JSON standardizzato, organizzati in endpoint (/register, /new_game, /add_request, ecc.) e gestiti attraverso un sistema di polling asincrono (controllo periodico del server), che consente al client di rispondere dinamicamente a notifiche dal server, come richieste di partita o accettazioni.

La schermata principale (HomePage) è progettata come un controller composito, con moduli separati che gestiscono la creazione dei widget, le azioni degli

utenti e la sincronizzazione delle richieste di gioco. Questo approccio favorisce la separazione delle responsabilità, rendendo il codice più facile da mantenere e ampliare.

Per quanto riguarda l'interfaccia, è stato scelto un tema scuro coerente, un layout responsivo basato su grid() e un sistema di scrolling personalizzato per liste dinamiche. Lo stato dell'applicazione è mantenuto in una struttura condivisa che persiste durante la navigazione tra le schermate, permettendo di gestire sessioni utente, ID giocatore, partite attive e simboli assegnati.

È stata prestata particolare attenzione alla gestione degli errori, con strategie di degradazione controllata e gestione delle disconnessioni.

Durante la fase di sviluppo dell'applicazione, è risultato fondamentale implementare sul lato client un meccanismo di server polling, incaricato di monitorare la socket di comunicazione con il server. Grazie a questo approccio, il client è in grado di ricevere messaggi in tempo reale senza interrompere o bloccare l'esecuzione dell'interfaccia grafica. Per gestire la comunicazione in modo asincrono e non bloccante, si è fatto uso della funzione select(), che permette di rilevare la presenza di dati pronti per la lettura sulla socket prima di effettuare l'operazione di ricezione.

4 Containerizzazione dell'applicazione

Per la nostra applicazione, abbiamo deciso di utilizzare Docker e Docker Compose come strumenti chiave per la containerizzazione e la gestione dei nostri ambienti di sviluppo e produzione. Questa scelta ci ha permesso di racchiudere l'applicazione e tutte le sue dipendenze in container isolati, assicurandoci un ambiente coerente e facilmente replicabile su qualsiasi macchina. Grazie a Docker, possiamo essere certi che il software funzioni allo stesso modo ovunque. Docker Compose, d'altra parte, rende molto più semplice l'orchestrazione di servizi multi-container (nel nostro caso server e frontend), definendo l'intera architettura in un unico file YAML. Questo approccio semplifica notevolmente la configurazione dell'ambiente di sviluppo, il testing e il deployment, riducendo il carico di lavoro e migliorando l'efficienza del ciclo di vita del software. Docker Compose genera inoltre un bridge di rete dedicato, denominato "tris-network", che abilita la comunicazione interna tra i diversi servizi containerizzati dell'applicazione (client e server), isolandoli dalla rete host e garantendo connettività privata e sicura.

Dato che l'applicazione include un client grafico sviluppato in Python con Tkinter, per il suo deployment e l'esecuzione in un ambiente Linux containerizzato, è necessario l'utilizzo del X Window System (X11) essendo che Docker, di default, non include un server grafico. Pertanto, per visualizzare l'interfaccia dell'applicazione dal container sul display dell'host (la macchina su cui Docker è in esecuzione), la configurazione prevede l'implementazione del X11 forwarding.

4.1 Gestione Multi-Client e Script di Utility

Essendo il progetto progettato per supportare un ambiente multi-client, abbiamo definito degli script Bash dedicati per semplificare la gestione delle istanze e delle operazioni Docker.

4.1.1 Script manage_clients.sh

Questo script bash fornisce funzionalità avanzate per la gestione multi-client, orchestrando l'avvio, lo stato, e la terminazione di più istanze client.

Di seguito i comandi disponibili:

- `./manage_clients.sh start [N]` : Avvia `N` client.
- `./manage_clients.sh status` : Visualizza lo stato di tutti i container client.
- `./manage_clients.sh logs` : Visualizza i log di tutti i container client.
- `./manage_clients.sh logs [name]` : Visualizza i log di un determinato container client tramite il suo nome.
- `./manage_clients.sh stop` : Ferma tutti i client.
- `./manage_clients.sh restart` : Riavvia il server.
- `./manage_clients.sh clean` : Esegue una pulizia completa dell'ambiente Docker (ferma e rimuove tutti i container, reti, volumi, etc.).

4.1.2 Script docker-run.sh

Questo script bash più semplice è stato creato per le operazioni di base di build e avvio dei componenti principali dell'applicazione.

Di seguiti o comandi disponibili:

- `./docker-run.sh build` : Costruisce le immagini Docker necessarie per l'applicazione.
- `./docker-run.sh server` : Avvia solo il container del server.
- `./docker-run.sh client` : Avvia un singolo container client.
- `./docker-run.sh all` : Avvia sia il server che un client.
- `./docker-run.sh stop` : Ferma tutti i container e i servizi.

4.2 Vantaggi Finali dell'Approccio Docker

L'adozione di Docker e Docker Compose per la nostra applicazione ha portato a numerosi vantaggi concreti:

1. **Sviluppo Accelerato**: Il setup dell'ambiente di sviluppo avviene in pochi secondi, riducendo drasticamente i tempi.
2. **Testing Semplificato**: La possibilità di avviare e gestire facilmente più client con un unico comando semplifica notevolmente i test .
3. **Deployment Consistente**: L'uso dei container garantisce che la stessa configurazione di sviluppo sia replicata fedelmente in qualsiasi ambiente di deployment.
4. **Scalabilità**: L'architettura permette di scalare facilmente da un singolo client a `N` client senza modifiche complesse al codice o alla configurazione.
5. **Isolamento**: I container forniscono un isolamento efficace, evitando conflitti tra le dipendenze dell'applicazione e il sistema operativo host.
6. **Debugging**: La capacità di accedere a log centralizzati e la possibilità di debugging remoto all'interno dei container migliorano l'efficienza nella risoluzione dei problemi.
7. **Manutenibilità**: La configurazione dell'ambiente è versionata e tracciabile tramite i file `Dockerfile` e `docker-compose.yml`, semplificando la manutenzione e gli aggiornamenti futuri.