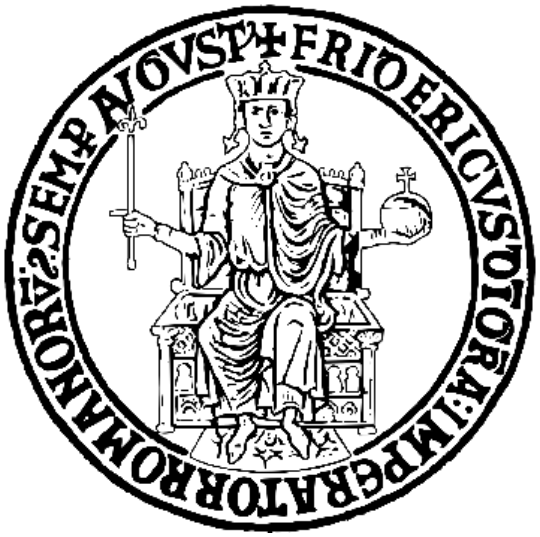

Corso Di Laurea In Informatica

Anno Accademico 2024/2025

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

Antonio Esposito

Matricola N86005375

antonio.esposito167@studenti.unina.it

Raffaele Giustiniani

Matricola N86003051

ra.giustiniani@studenti.unina.it

Indice

Capitolo 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

```
tris-lso/
├── Server/
│   ├── Dockerfile
│   ├── main.c
│   ├── funzioni.c
│   ├── funzioni.h
│   ├── strutturedati.h
│   └── Makefile
├── Client/
│   ├── Dockerfile
│   ├── main.c
│   ├── funzioni.c
│   ├── funzioni.h
│   ├── strutturedati.h
│   └── Makefile
├── avviowindows.ps1
├── avviounix.sh
└── docker-compose.yml
```

Capitolo 2

Progettazione

Come primo passo sono stati individuati i macro-elementi principali del sistema da definire: i giocatori e le partite. Analizzandone i possibili stati e fasi del loro "ciclo di vita" è stato possibile produrre uno scheletro delle principali funzionalità e strutture dati utili, suddividendone le responsabilità tra Client e Server.

2.1 Funzionalità principali

Innanzitutto, il server si occupa di creare un thread esclusivo per ogni client, associato a sua volta ad un unico giocatore, per poi tornare in ascolto per altre connessioni.

Ogni thread, su un livello più astratto, gestisce le 3 fasi del ciclo di vita di un giocatore: registrazione, attesa e partita. Mentre la prima viene eseguita una sola volta per giocatore, le altre due si alternano fino all'eventuale disconnessione del client. Per la fase di attesa abbiamo cercato di replicare ciò che fa la maggior parte dei giochi online: introdurre il concetto di "lobby", una stanza dove i giocatori che non sono impegnati in una partita possono decidere cosa fare.

Registrazione

Non utilizzando account ma solo ospiti temporanei, il sistema si limita a chiedere un username, che dopo un controllo di unicità userà sia come identificativo del giocatore che come nome dell'eventuale partita di cui è proprietario.

Lobby

Conclusa la registrazione, il giocatore entra in lobby e riceve una lista di tutte le partite attive del sistema e le relative informazioni (proprietario, stato, etc...). Queste informazioni vengono aggiornate in tempo reale ogni volta che vengono modificate da altri thread finché il giocatore resta in lobby. Inoltre, ogni volta che un nuovo giocatore si registra ed entra per la prima volta in lobby, tutti gli altri ricevono il messaggio "[nome_giocatore] è appena entrato in lobby!". Mentre è in lobby, un giocatore può decidere se unirsi ad una partita in attesa, crearne una nuova o uscire dal gioco.

Partita

Per entrare in una partita si può crearne una e attendere che un avversario si unisca, oppure richiedere di unirsi ad una in attesa. Durante la partita ciascun giocatore ha la possibilità di arrendersi e tornare in lobby ad ogni turno. Finita la partita, se il vincitore non era originariamente il proprietario della partita, lo diventa, e può decidere se aspettare un nuovo avversario o tornare in lobby, mentre lo sconfitto è costretto a tornare in lobby. In caso di pareggio, entrambi i giocatori possono decidere se giocare un altro round scambiandosi il primo turno, ed in caso di risposta negativa di almeno uno dei due, il proprietario può cercare un nuovo sfidante, mentre l'avversario torna in lobby.

2.2 Responsabilità Server-side

Come detto precedentemente, appena un nuovo giocatore si connette, viene creato un nuovo thread ad egli associato. È utile tenere a mente che questo è l'unico tipo di thread usato dal server, quindi in tale ambito si può immaginare una terna giocatore-client-thread in ordine discendente di livello di astrazione. Il processo server ha il compito di gestire i dati di giocatori e partite, a tale scopo questi sono globali ed accessibili ad ogni thread (in modo controllato con l'uso di mutex per operazioni atomiche) il quale effettua controlli e modifiche su di essi. Ciascun thread gestisce la registrazione e la comunicazione col giocatore-client per tutta la durata dell'utilizzo del servizio, cancellandosi non appena il giocatore si disconnette volontariamente o dopo un errore di rete.

In particolare, in fase di registrazione, il thread ha il compito di ricevere il nome del giocatore, effettuare controlli sui giocatori esistenti, ed allocare memoria per il nuovo giocatore.
In fase di lobby, il thread si occupa dell'invio di statistiche, informazioni sulle partite attive e dei loro aggiornamenti (l'implementazione verrà discussa nella sezione di gestione dei thread, essendo strettamente collegata ai signal handler). Una responsabilità importante in questa fase è quella di far sapere al client quando il giocatore sta entrando in partita.

Durante la partita, la comunicazione tra due giocatori viene gestita dal thread del proprietario della partita, mentre il thread dell'avversario resta in attesa fino al ritorno in lobby (anche in questo caso l'implementazione verrà discussa nella sezione di gestione dei thread). Per essere il più leggero possibile, il server non si occupa di gestire la partita in sé, ma solo di ritrasmettere le giocate da un client all'altro (senza effettuare controlli), controllare l'esito della partita ricevuto dai client ed eventuali errori di rete. Appena la partita finisce il thread del proprietario aggiorna le statistiche dei giocatori e in base all'esito decide come comportarsi, comunicando al client l'eventuale ritorno in lobby o la rivincita.

2.3 Responsabilità Client-side

A differenza del server, il processo client può essere diviso in due fasi principali, una di ascolto e scrittura (la quale ingloba le fasi di registrazione e lobby), l'altra che gestisce la partita. Per tutta la durata del processo, il client si occupa di fare controlli preliminari sulla validità degli input, in modo che il server riceva sempre messaggi validi.

Il processo client inizia creando un thread scrittore, in modo da restare in ascolto passivo per eventuali aggiornamenti dal server durante la fase di lobby, dando allo stesso tempo la possibilità di ricevere input e quindi mandare messaggi al server.

All'inizio di una partita, ossia alla ricezione di una stringa speciale dal server, il client entra nella seconda fase, terminando il thread scrittore, poiché durante la partita gli input possano essere gestiti in modo sequenziale dal thread principale.

Il client si occupa della gestione della propria partita, modificando e mostrando al giocatore la griglia di gioco. Durante una partita, se è il proprio turno, il codice del client prende in input la giocata, aggiorna la griglia, definisce l'esito e lo invia al server insieme alla giocata stessa. Se la partita è ancora in corso, riceve la giocata dell'avversario, aggiorna la griglia e ricalcola l'esito. A fine partita, a seconda dell'esito, torna nella fase di ascolto e lettura, ricreando un thread scrittore, oppure comunica col server per fare una rivincita o cercare un nuovo avversario.

Capitolo 3

Implementazione

In questo capitolo si discutono alcune specifiche sull'implementazione del sistema progettato, quali l'implementazione di funzioni e strutture dati, la gestione di thread e segnali, la gestione degli errori e la tipologia di socket usate per la comunicazione.

3.1 Impostazioni e comunicazione Socket

Le socket sono di tipo Ipv4, con protocollo di rete TCP. Per collegarsi al server il client utilizza l'indirizzo ip del localhost, cioè 127.0.0.1 (nel capitolo 4 è spiegato meglio come funziona la connessione). Il server si mette in ascolto sulla porta 8080, mentre la socket del client non viene inizializzata in modo particolare e non viene usata la funzione bind(), lasciando al sistema operativo il compito di assegnare automaticamente una porta disponibile al client.

Utilizzando la funzione setsockopt(), è stato possibile aggiungere delle funzionalità aggiuntive alle socket per migliorare il funzionamento dell'applicazione. Alla socket del server è stata abilitata l'opzione SO_REUSEADDR, evitando di generare l'errore in fase di binding "address already in use", che si verifica quando si cerca di creare una socket che utilizza una porta sulla quale è o era recentemente in ascolto un altro processo. Questo permette di riavviare velocemente il server senza dover aspettare che il SO liberi la porta e ci è stato estremamente utile in fase di testing.

Alla socket client, invece, è stata data l'opzione TCP_NODELAY, che permette di disattivare il buffering dei dati su una socket TCP prima dell'invio (algoritmo di Nagle), il che permette nel caso di applicazioni che si scambiano ridotte quantità di dati come la nostra di ridurre la latenza e consentire invio e ricezione di informazione più rapidi.

Tramite le opzioni SO_SNDTIMEO e SO_RCVTIMEO è stato inoltre impossibile impostare un "timer" di 300 secondi sia sulle send che sulle recv: se il processo si blocca su una di queste 2 system call per 5 minuti il timer scade, le funzioni restituiscono –1 e l'error number viene impostato di conseguenza, facendo automaticamente chiudere il processo dopo aver stampato all'utente un messaggio di disconnessione per inattività (anche se è da notare che un evento del genere è praticamente impossibile accada sulle recv nel nostro caso).

Infine, le send del server sono impostate col flag MSG_NOSIGNAL, che evita l'invio del segnale SIGPIPE quando il lato client della socket è stato chiuso ordinatamente o a causa di un errore, questo perché di default il segnale in questione avrebbe fatto crashare l'intero processo server. Per gestire il problema avremmo anche potuto fare in modo che il server ignorasse il segnale SIGPIPE tramite l'istruzione signal(SIGPIPE, SIG_IGN), ma abbiamo preferito gestire il problema direttamente al livello delle socket.

3.2 Strutture dati

Il client non fa uso di particolari strutture dati in quanto non ha essenzialmente nulla da gestire ed organizzare oltre alla griglia di gioco (una semplice matrice 3x3).

Il server utilizza due liste concatenate protette da dei mutex, una per gestire i giocatori e una per le partite. Ogni nodo giocatore viene allocato dal thread che se ne occupa, inizializzato ed aggiunto in lista in fase di registrazione, mentre un nodo partita viene allocato ogni volta che un giocatore decide di creare una nuova partita e cancellato quando quest'ultima termina definitivamente (ossia senza rivincite).

```c
struct nodo_giocatore
{
    char nome[MAXPLAYER];
    enum stato_giocatore stato;
    pthread_cond_t stato_cv;
    pthread_mutex_t stato_mutex;
    bool campione;
    unsigned int vittorie;
    unsigned int sconfitte;
    unsigned int pareggi;
    pthread_t tid_giocatore;
    int sd_giocatore;
    struct nodo_giocatore *next_node;
};
```

```c
struct nodo_partita
{
    char proprietario[MAXPLAYER];
    int sd_proprietario;
    char avversario[MAXPLAYER];
    int sd_avversario;
    enum stato_partita stato;
    bool richiesta_unione;
    pthread_cond_t stato_cv;
    pthread_mutex_t stato_mutex;
    struct nodo_partita *next_node;
};
```

Sorvolando sul ruolo ovvio delle altre variabili, spieghiamo brevemente il significato di alcune più oscure:

- bool campione: indica se il giocatore ha vinto la sua ultima partita, utile a facilitare i controlli su chi deve diventare proprietario della prossima partita e chi deve tornare in lobby.
- bool richiesta_unione: introdotta per evitare che più giocatori potessero richiedere di unirsi alla stessa partita, prima che il proprietario abbia risposto al primo sfidante. Viene impostata a true quando il proprietario della partita riceve una richiesta, poi a false appena risponde.

Le variabili di stato, assieme alle variabili di condizione ed i mutex associati, verranno discussi nella sezione seguente.

3.3 Gestione dei thread

Tutti i thread, sia client che server, vengono creati in modalità detatched, poiché i processi che li generano non hanno bisogno di conoscerne lo stato di terminazione.

La funzionalità server di aggiornamento di tutti i giocatori in lobby, quando le partite cambiano stato, vengono create o cancellate, è stata implementata sfruttando il segnale personalizzabile SIGUSR1. In particolare, i thread che fanno queste modifiche chiamano la funzione segnala_cambiamento_partite(), che a sua volta manda un segnale con pthread_kill(tid, SIGUSR1) a tutti i giocatori in lobby escluso il mittente del segnale. L'handler associato a SIGUSR1 è invia_partite(), che invia al client (sempre in ascolto passivo mentre è in lobby) la lista di partite attive.

In maniera totalmente analoga, alla registrazione di un nuovo giocatore viene inviato un segnale SIGUSR2 a tutti i giocatori in lobby, il cui handler ne notifica i client.
Il client invece usa il segnale SIGUSR1 per terminare il thread scrittore.

Quando una partita ha inizio, lo sfidante si blocca sulla propria variabile di condizione stato_cv, associata al suo stato, tramite la funzione pthread_cond_wait(&stato_cv, &stato_mutex), e viene sbloccato solo alla fine della partita dalla funzione complementare pthread_cond_signal(&stato_cv) fatta dal thread del proprietario quando la partita finisce e lo stato dell'avversario torna ad avere stato IN_LOBBY (a meno che non venga chiesta una rivincita), oppure dall'error handler, ossia la funzione che si occupa di gestire gli errori di rete e le disconnessioni improvvise.
Analogamente, il thread del proprietario si blocca sulla variabile di stato partita mentre è in attesa di un avversario (finché il thread di quest'ultimo non la rende IN_CORSO) con una timedwait, per verificare periodicamente che il client non si sia disconnesso lasciando il mutex lockato all'infinito.

Oltre ai mutex associati alle variabili condizionali di ciascun giocatore e partita, vengono utilizzati altri due mutex, uno per lista, che vengono utilizzati per atomizzare le sezioni critiche che operano sulle rispettive liste. Da notare che questi mutex, a differenza di quelli delle variabili condizionali, vengono allocati staticamente.

3.4 Gestione degli errori

Ogni chiamata a send e recv gestisce eventuali errori chiamando error_handler().
Nel client, questa funzione si limita a stampare un messaggio a seconda del tipo di errore (per inattività o generico), chiude la socket se serve e termina il processo.

Nel server, controlla se il giocatore disconnesso era in partita, se era il proprietario o l'avversario, e a seconda dei casi si occupa di assegnare la vittoria, cambiare lo stato dell'avversario (invocando anche pthread_cond_signal(&stato_cv)), avvisare l'avversario della disconnessione (inviando il flag di errore spiegato sotto) e/o cancellare il nodo partita dalla lista. Dopodiché chiama pthread_kill(tid_giocatore, SIGALRM), il cui handler associato cancella il nodo del giocatore dalla lista, ne chiude la socket e termina il thread.

Normalmente il server invia ai client un flag utilizzando la costante NOERROR prima di inviare ogni giocata, mentre se si verifica un errore a causa dell'avversario, l'error handler del server invia il flag ERROR.

Prima di inviare e ricevere giocate, il client riceve anche il flag di errore dal server, e se quest'ultimo corrisponde ad ERROR significa che l'avversario si è disconnesso, quindi la partita termina automaticamente e viene visualizzato un messaggio di vittoria a tavolino.

L'handler per il segnale SIGTERM (inviato da docker per stoppare i container) chiude le socket e fa terminare i processi, viene usato per velocizzare la chiusura che altrimenti impiega intorno ai 10 secondi per container.

Capitolo 4

Docker

Nel nostro progetto sono stati creati due Dockerfile: uno per creare il container che definisce il servizio Server, uno per il servizio Client, ed un file docker-compose.yml, che consente di far interagire tra loro i container in modo semplice ed internamente alla rete docker, senza dover obbligatoriamente esporre una porta dell'host per consentire la connessione dei client al server.

4.1 Dockerfile

I dockerfile del client e del server sono del tutto simili tra di loro ed il più semplice possibili. Di seguito la lista delle istruzioni contenute nei file ed il loro significato:

FROM ubuntu:latest

L'istruzione FROM indica l'immagine di partenza dalla quale costruire il resto dell'immagine, inizialmente avevamo deciso di usare Arch Linux perché estremamente leggero e veloce, ma alla fine la scelta è ricaduta su Ubuntu perché molto più stabile e facile da usare, al costo di qualche MB in più nel peso finale dell'immagine.

RUN apt-get update && apt-get install –y gcc make && rm –rf /var/lib/apt/lists/*

L'istruzione RUN fa eseguire a docker un comando all'interno dell'immagine. In questo caso, aggiorna i pacchetti disponibili, installa il compilatore gcc ed il comando make. Finita l'installazione, cancella la cache di apt per ridurre il peso totale dell'immagine.

WORKDIR /app

Imposta una directory di lavoro chiamata "app".

COPY . .

Copia tutto il contenuto della cartella dove è contenuto il Dockerfile nella nuova directory di lavoro.

RUN make

Utilizza il comando make, che a sua volta sfrutta il Makefile nella directory di lavoro per compilare il codice e creare un eseguibile.

CMD ["./server"] e CMD ["./client"]

L'istruzione CMD definisce il primo comando da eseguire dopo l'avvio del container, quindi ./server e ./client per i rispettivi eseguibili.

4.2 Docker Compose

Il file docker-compose.yml si trova nella directory principale “tris-lso” e permette ai container client e server di comunicare tra loro in maniera efficiente e soprattutto all’interno della rete docker, senza il bisogno di sfruttare delle porte della macchina host per la connessione.

In particolare, nel file Compose vengono definiti quelli che docker chiama “servizi”: il servizio server ed il servizio client, e viene ad entrambi dato un contesto di build, cioè un’indicazione su dove si trova il Dockerfile che costruisce l’immagine del servizio. Al server basta questo per funzionare, mentre il servizio client necessita di altre istruzioni per potersi connettere al server e poter prendere input da un utente esterno.

L’istruzione “depends_on : server” garantisce che il client aspetti che il server sia in esecuzione prima di avviarsi, mentre le istruzioni "stdin_open: true” e “tty: true” servono ad “aprire” lo standard input del processo (che docker chiude di default) e a dare la possibilità di “attaccare” (col comando docker attach) un terminale al container.

Infine, l’istruzione “network_mode : “container:server”” serve ad indicare al client la modalità di rete da utilizzare, il parametro “container:server” indica la rete usata dal container server, permettendo al client di collegarsi a quest’ultimo tramite l’indirizzo ip del localhost (127.0.0.1). Un’altra opzione sarebbe stata quella di creare un network personalizzato ed aggiungere i due servizi ad esso, ma è più indicata quando serve che un grande numero di container interagiscano tra loro. L’istruzione network_mode è più “elegante” e più che sufficiente per i nostri scopi.

Capitolo 5

Note d’uso - script d’avvio

L’avvio dell’applicazione è automatizzato da uno script PowerShell per Windows (che può essere usato anche su Linux e macOS dato che Powershell è multipiattaforma) e da uno script Bash per sistemi Unix-like, in particolare macOS e le distribuzioni Linux che utilizzano i più comuni emulatore di terminale (gnome, konsole, xfce4 e xterm).

Gli script di avvio utilizzano il comando “docker-compose up --scale client=n”, facendo partire docker compose che a sua volta avvierà un container server ed n container client, secondo le istruzioni scritte nei Dockerfile e nel file docker-compose. Mentre docker compose viene eseguito, gli script vengono bloccati in un piccolo ciclo dove periodicamente (ogni 2 secondi) controllano che tutti i container avviati siano in esecuzione, per poi entrare in un ciclo for dove viene usato il comando “docker attach tris-lso-client-i", che collega un terminale della macchina host ad un container client, permettendo all’utente finale di gestire i client in modo semplice e comodo senza preoccuparsi di cercare di capire come funziona l’interfaccia docker.