1 Introduzione

Nella progettazione dell'applicazione distribuita alla quale questa documentazione è allegata mi sono prefissato una serie di obiettivi fondamentali, che mi permettessero di produrre un buon progetto e di imparare il più possibile dalla sua realizzazione. Il primo è l'aderenza alla traccia e l'efficienza nelle comunicazioni sulla rete. Il secondo è l'ottimizzazione delle operazioni effettuate in locale, sia a livello di strutture dati che di generazione di nuovi processi. Il terzo, infine, è la qualità del codice: ho provato a scrivere sorgenti che fossero il più possibile modulari, ben commentati, e soprattutto che garantissero la più ampia flessibilità nell'utilizzo. Vado adesso a discutere le varie scelte implementative, rimandando per ulteriori approfondimenti ai commenti nel codice e, da ultimo, alla discussione in sede d'esame.

2 Comunicazioni sulla rete

L'applicazione utilizza esclusivamente il protocollo TCP: è infatti un'applicazione loss-sensitive, che può funzionare solo se i pacchetti vengono ricevuti integri e nell'ordine corretto. Lo scenario in cui l'applicazione opera, poi, si contraddistingue per l'assenza di interazioni tra client: anche se le classifiche sono pubbliche, ogni utente vive un'esperienza di gioco eminentemente individuale, e personalizzabile in base alle proprie scelte (si pensi al nickname, o al tema scelto). Inoltre, le risposte possono essere date senza vincoli temporali, e pertanto i vari client rispondono in maniera reciprocamente asincrona. Alla luce di tali considerazioni, un server concorrente risulta essere il più adatto: permette, infatti, di dedicare a ciascun client un proprio processo server, parallelizzando l'accesso a domande, risposte e nomi dei temi, e quindi efficientando significativamente l'esperienza di gioco. Come illustrato di seguito, si è scelto di ottimizzare ulteriormente il codice tramite l'impiego dei thread.
Per progettare una struttura dei pacchetti che fosse al contempo sicura e leggera, si è analizzata la struttura del gioco come una macchina a stati finiti, come riportato in figura. Dal diagramma risulta evidente che è possibile evitare di definire – e trasmettere ogni volta – un tipo del messaggio, a condizione che non sia possibile definire una risposta denominata "show score" o "endquiz" e che client e server effettuino adeguati controlli sugli input dell'utente. Poiché queste condizioni non pregiudicano la flessibilità e la qualità dell'applicazione, si è scelto di adottarle riducendo le dimensioni dei messaggi.
Dal momento che la maggior parte dei dati inviati sulla rete sono stringhe di lunghezza variabile (domande, risposte, nickname, nomi dei temi) si è scelto di non adottare formati predefiniti, che avrebbero comportato un importante spreco di risorse a ogni trasmissione. Per lo stesso motivo, si sono utilizzati effettivamente protocolli di tipo text: la compressione sarebbe stata limitata ai soli valori interi, che costituiscono una percentuale minima dei dati trasmessi, e non avrebbe giustificato l'overhead dovuto alla serializzazione delle strutture dati. Pertanto, ogni trasmissione è preceduta dall'invio di un intero contenente la lunghezza del pacchetto. Sempre con l'intento di minimizzare i dati trasmessi, la marca di fine stringa non viaggia sul canale: viene rimossa in trasmissione e aggiunta in ricezione. Da ultimo, si è dedicata molta cura alla gestione delle terminazioni della connessione: oltre alla loro rilevazione in fase di ricezione, approfondita nel corso, si è

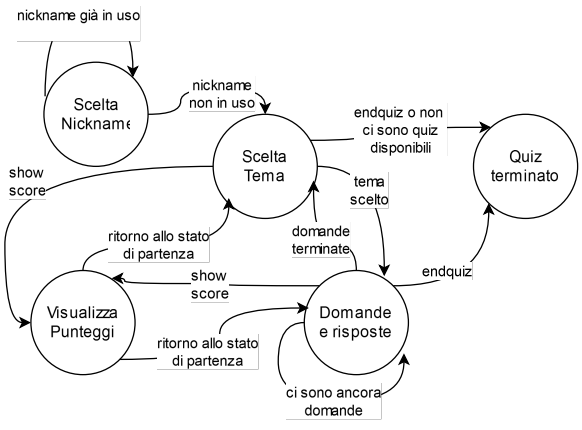

Figure 1: Struttura del gioco vista come Macchina a Stati

implementata una rilevazione in fase di invio, che intercetta il segnale SIGPIPE e lo associa a un'apposita routine di gestione. È possibile terminare il server correttamente, chiudendo la connessione, digitando il carattere 'q' e premendo invio.

3 Processi e Strutture dati

Dal momento che l'applicazione fornisce servizi al pubblico, è ragionevole aspettarsi un numero considerevole di connessioni attive contemporaneamente, e conseguentemente di dati memorizzati nel server. Pertanto, ho prestato particolare attenzione all'ottimizzazione delle prestazioni del server. Il primo aspetto che ho analizzato in questa prospettiva è stato l'overhead dovuto ai cambi di contesto: dal momento che a ogni connessione corrisponde un processo, il sistema può trovarsi a impiegare una parte considerevole del suo tempo in cambi di contesto tra processi server, nonostante questi condividano il codice e le strutture dati. Per abbattere l'overhead e migliorare così le prestazioni, ho scelto di utilizzare i processi leggeri, o thread, che condividono la stessa immagine di processo e pertanto possono alternarsi sui processori in modo più efficiente.
Il secondo bottleneck individuato nelle prestazioni del server è stato invece quello delle strutture dati, e in particolare del database degli utenti e della classifica. Queste strutture sono accedute molto frequentemente: si pensi alla loro stampa a video periodica nella dashboard del server o all'aggiornamento dei punteggi degli utenti. Pertanto, era cruciale efficientare le operazioni sui dati, con particolare riferimento all'inserimento e alla stampa in ordine. A tale scopo si è scelto di utilizzare gli alberi binari di ricerca, che, denominato n il numero dei nodi, permettono un inserimento in O(log(n)) e una stampa ordinata in O(n) tramite visita in-order. Nello specifico, sono state implementate due tipologie alberi: quello degli utenti, ordinato in ordine alfabetico crescente sui nickname, e quello dei punteggi per ogni tema, ordinato per punteggio decrescente e, in subordine, in ordine alfabetico crescente sui nickname.
Da ultimo, si è ritenuto opportuno snellire quanto più possibile le operazioni di parsing iniziale dei dati dei quiz. A tale scopo è stato definito un apposito formato di file .quiz che, tramite l'utilizzo di caratteri speciali come delimitatori, può essere letto efficientemente da parte di funzioni dedicate.

4 Il codice

Come anticipato in incipit, nella progettazione dell'applicazione si sono sempre presi a riferimento i principi dell'ingegneria del software, perseguendo al massimo la modularità e la flessibilità.
Il codice a disposizione di client e server si articola in 5 moduli, il cui scopo è dettagliato nei commenti al codice, ognuno con una propria interfaccia e la relativa implementazione. **Le indicazioni sulla compilazione del codice sono contenute nel Makefile allegato al progetto. Il progetto può essere compilato e lanciato in maniera automatica tramite il file start.sh.**
Infine, si è scelto di privilegiare la massima flessibilità del codice, anche quando perseguirla significava complicare notevolmente l'implementazione. Il numero dei temi, ad esempio, non è contenuto nel codice, ma valutato a runtime in base al numero di file .quiz presenti nella cartella dedicata e al contenuto del file indice.quiz, fondamentale per l'accesso efficiente a domande e risposte. Inoltre, tutti i parametri che determinano il funzionamento dell'applicazione sono configurabili nell'interfaccia params.h. Tra questi rientrano, tra gli altri, il tempo di aggiornamento del pannello di controllo, la lunghezza dei buffer, le dimensioni dei quiz, i codici di errore restituiti e i comandi che possono essere lanciati dall'utente, oltre ovviamente ai dati relativi alla connessione, come l'indirizzo IP, la porta del server e il backlog.