UNIVERSITÀ DI PISA

Progetto Reti Informatiche

Corso di laurea in Ingegneria Informatica
Anno Accademico 2024-2025

Matteo Bellini

Introduzione

L'applicazione distribuita è stata sviluppata in modo che sia il più semplice possibile e che siano rispettati i requisiti richiesti dalla traccia. Il codice è stato scritto con l'obiettivo di essere il più snello e leggibile possibile, anche grazie alla presenza di commenti in molte parti di questo. Client e Server utilizzano alcune costanti definite nel file *parametri.h*, come: la lunghezza massima dei buffer, la lunghezza massima di un input, l'indirizzo del server, la dimensione del backlog nel server e il numero di domande per ogni quiz. I due applicativi inoltre utilizzano alcune funzioni di utilità dichiarate nel file *utils.h*. Il codice è organizzato in cartelle:

- la cartella *dati* contiene un file per ogni quiz ed un file indice;
- la cartella *include* contiene le dichiarazioni delle funzioni e delle strutture dati utilizzate;
- la cartella *modules* contiene le definizioni delle funzioni utilizzate.

Per facilitare la compilazione del codice è stato creato un *makefile* e l'operazione può essere effettuata lanciando il comando *make* nella shell.

Server

Nella realizzazione del server si è optato per l'utilizzo di un server multithread che presenta un thread principale, il quale si occupa di accettare le nuove connessioni dei giocatori, e un thread per ogni giocatore connesso. Si è deciso di utilizzare questa tipologia di server vista la necessità di dover stilare delle classifiche per memorizzare i punteggi dei vari utenti e con l'utilizzo dei thread questa operazione risulta semplificata in quanto questi condividono le strutture dati globali. Un altro motivo per cui si è scelto di implementare un approccio multithread è il minor overhead nella creazione e nella distruzione dei processi leggeri rispetto ai processi pesanti.

Strutture Dati

Le strutture dati implementate sono state definite con l'obiettivo di ottenere un'applicazione semplice ma comunque efficiente.

Per la gestione dei quiz è stata definita la *struct quiz*, che memorizza al suo interno il numero di temi definiti ed un array di temi. Sono state poi definite le strutture dati *tema*, *domanda* e *risposta* necessarie per caricare in memoria le informazioni relative ai quiz. Queste strutture dati vengono popolate all'avvio del server e non vengono mai modificate durante tutta la sua esecuzione. Si è deciso di utilizzare queste strutture per evitare di dover effettuare numerosi accessi ai file contenenti i dati dei vari quiz, che avrebbe portato ad ottenere un'applicazione molto più inefficiente.

Per la gestione degli utenti registrati al servizio è stata definita la struttura dati *utenti*; questa al suo interno memorizza il numero di utenti registrati e una lista di elementi di tipo *struct utente*. Quest'ultima struttura contiene al suo interno le informazioni relative ad un utente registrato come l'username, i temi completati e quelli giocati. All'interno di questa struttura è presente il campo *bool endquiz* e nel caso in cui questo abbia valore true l'utente ha lanciato il comando *endquiz* ed è da considerarsi come inattivo. È stato necessario definire anche un

semaforo di mutua esclusione per l'accesso alla struttura dati utenti in quanto questa è condivisa dai thread.

Per la gestione dei punteggi degli utenti nei vari temi è stata definita la struttura dati *punteggio_tema* contenente al suo interno una lista di elementi del tipo *struct punteggio*. La struttura punteggio memorizza al suo interno il nome di un giocatore, il relativo punteggio e un puntatore all'elemento successivo. Anche in questo caso è stato necessario definire un semaforo di mutua esclusione visto l'accesso concorrente dei vari thread per l'aggiornamento delle classifiche. Si è deciso di definire delle strutture per la gestione dei punteggi invece di memorizzare questi dati direttamente all'interno della *struct utente* per ottenere una gestione più chiara e semplice. L'utilizzo delle liste inoltre permette di gestire l'ordinamento in maniera efficiente con complessità pari ad O(n).

Comunicazione tra Client e Server

Per la comunicazione tra Client e Server si è deciso di utilizzare il protocollo TCP in quanto per garantire il corretto funzionamento dell'applicazione è necessario che il trasferimento dei dati sia affidabile.
Come modalità di scambio dei dati si è utilizzato il tipo text, scelta molto condizionata dalla natura dei dati da scambiare tra il client e il server. Per ridurre la quantità di dati scambiati si è deciso di comunicare la dimensione delle stringhe prima di inviarle al destinatario.

Il client dopo essersi registrato presso il server riceverà la lista dei temi disponibili e poi terrà traccia di quelli a cui il giocatore ha già giocato in modo che questo non possa sceglierli nuovamente.

Per la gestione di eventuali errori o disconnessioni durante le comunicazioni sono state definite le funzioni gestoreErroriSend() e gestoreErroriRecv() sia per il client che per il server. Nel caso di recv() si verifica la disconnessione dell'interlocutore quando il valore restituito è 0 o -1 ed errno = ECONNRESET. Viene considerato errore anche il caso in cui il valore restituito è diverso da numero di byte che dovevano essere inviati.
Nel caso di send() si verifica un errore quando il valore restituito è diverso dal numero di byte da inviare; in particolare se la chiamata restituisce -1 ed errno = ECONNRESET oppure EPIPE si è verificata la disconnessione dell'interlocutore.
Nel caso di send() è stato necessario utilizzare il flag MSG_NOSIGNAL in ogni chiamata per evitare che la chiamata su un socket che potrebbe essere stato chiuso generi il segnale SIGPIPE che causerebbe la terminazione del programma.

In caso di disconnessione di un client il server deve eliminare il giocatore dalle classifiche dei temi e dalle liste dei temi completati, ed inoltre deve segnare il giocatore come inattivo. Nel client il lancio del comando *endquiz* porta alla chiusura del socket e all'avvio di una nuova sessione. La chiusura del socket non viene interpretata come un errore dal server in quanto prima di effettuare le operazioni di chiusura il client comunica al server che il comando è stato digitato.