UNIVERSITÀ DI PISA

Progetto di Reti Informatiche

Corso di Laurea in Ingegneria Informatica

Anno Accademico 2024-2025

Valerio Cannalire

Introduzione e file di progetto

Lo sviluppo dell'applicazione si è svolto con il principale obiettivo di mantenere questa il più semplice possibile: codice pulito, snello e facilmente leggibile, comunicazioni e strutture dati semplici e convenzionali. Per la compilazione dell'applicazione è presente un semplice makefile che permette di compilare tutti i file di progetto scrivendo il comando *make* nella shell. Client e server utilizzano alcune costanti numeriche comuni presenti nel file *costanti.h* che definiscono le lunghezze dei messaggi e le dimensioni dei relativi buffer. Sono state definite due librerie contenenti le funzioni usate da client e server: *libclient* e *libserver*. Per ognuna di esse sono presenti i rispettivi header e source file. All'interno della cartella *domande&risposte* sono presenti un file contenente le domande e uno contenente le risposte per ogni tema del gioco.

Tipologia di server

Al fine di servire più utenti contemporaneamente con rapidità ed efficienza si è da subito optato per una tipologia di server che permettesse la gestione di più flussi di esecuzione. L'idea di base era quella di un flusso principale che si occupasse di accettare le nuove richieste dei client e di delegare poi la gestione delle comunicazioni con essi a dei flussi di esecuzione secondaria. Vista poi la necessità di stilare una classifica comune a tutti i giocatori e di realizzare altre strutture dati condivise tra tutti i flussi si è definitivamente scelto un server multithread poiché offre la possibilità di condividere porzioni di memoria tra tutti i thread.

Strutture dati

Le decisioni riguardanti le strutture dati che implementano la logica e l'interfaccia del server sono state prese dopo la valutazione di più opzioni, privilegiando sempre l'obiettivo di creare un'applicazione il più semplice possibile ma non troppo inefficiente.

La verifica dell'univocità del nickname richiede necessariamente l'implementazione di una lista poiché il server deve memorizzare nel suo runtime i nomi di tutti i giocatori che hanno partecipato al quiz, e il loro numero non è noto a priori. Prime riflessioni avevano ritenuto necessaria la definizione di una *struct giocatore* per salvare le informazioni relative a un giocatore all'interno del server. Invece di definire due diverse strutture dati, una per la sola verifica del nickname e una per la gestione delle altre informazioni, si è deciso di collassare tutto nella *struct giocatore* e di aggiungere a essa un campo *bool attivo* per disitinguere gli utenti in gioco da quelli non più attivi. Durante lo sviluppo dell'applicazione, tuttavia si è ritenuto più efficiente delegare più compiti al lato client di quanti previsti inizialmente, rendendo di fatto inutili per le specifiche richieste tutti i campi eccetto *nickname* all'interno della *struct giocatore*. Si è deciso comunque di mantenere tale struttura dati poiché ritenuta coerente con le possibilità di espansione delle funzionalità dell'applicazione (ad esempio classifiche dei giocatori non più attivi).

Per gestire le classifiche, gli elenchi dei giocatori che hanno completato un tema, e i path dei file contenuti all'interno della cartella *domande&risposte*, è stata realizzata una struttura *temi*. La scelta della struttura dati per la realizzazione delle classifiche è stata principalmente condizionata dalla necessità di riordinamento in seguito all'incremento di un punteggio. Considerato questo aspetto la scelta è ricaduta su array dinamici sui quali è quindi possibile usare la *qsort()*. Nel momento in cui un nuovo giocatore sceglie un tema, si alloca un nuovo vettore per la classifica con un elemento in più, si copia l'array precedente e si aggiunge in fondo il nuovo giocatore. La precedente classifica viene poi deallocata. Analogamente quando un giocatore si disconnette viene allocato un nuovo array con un elemento in meno, si ricopiano tutti i punteggi eccetto quello del giocatore appena disconnesso, e poi si dealloca la precedente classifica. Gli elenchi dei giocatori che hanno completato il tema, non essendo soggetti a riordinamento sono implementati come semplici liste. Al fine di evitare parsing complessi, nei file all'interno della cartella *domande&risposte* ogni domanda/risposta occupa una riga, per cui è possibile effettuarne una lettura con una semplice *fgets*.

All'interno del client è presente un'unica struttura dati *temi* profondamente diversa rispetto alla sua controparte nel server. L'intento di tale struttura è infatti quello di gestire lato client l'elenco dei temi disponibili, mostrando a esso solo quelli che deve ancora svolgere, senza che ci sia il bisogno di interfacciarsi col server.

Comunicazione

Affinché l'applicazione funzioni correttamente, è fondamentale che la trasmissione dei pacchetti sia affidabile. Conseguentemente la scelta del protocollo di trasporto è ricaduta sul TCP.

Data la natura dei dati trattati, la modalità di scambio è prevalentemente di tipo text con un paio di eccezioni discusse in seguito. Al fine di ridurre la quantità di scambi si è preferito non comunicare la lunghezza prima dell'invio del messaggio sebbene le stringhe possano avere dimensioni molto variabili. Si è quindi adottato un compromesso basato sulla lunghezza predefinita dei messaggi per mezzo delle costanti definite in *costanti.h*. La semplicità dell'applicazione, infatti prevede lo scambio di messaggi dalla dimensione piuttosto esigua, per cui si è ritenuto trascurabile il costo dovuto all'invio di un numero maggiore di byte. Siccome sia client che server operano con funzioni della libreria *string.h* basate sulla presenza del terminatore di stringa, e gli input del giocatore avvengono sempre in sicurezza per mezzo della funzione *inserimento()* definita all'interno di *libclient*, non si incorre in problemi dovuti alla ricezione di un numero maggiore di byte. Per trasparenza si precisa che la funzione *inserimento()* è stata generata con l'ausilio di intelligenze artificiali (con *estrema* difficoltà da parte loro).

Solo nel caso di *show score* si è deciso di inviare la dimensione prima della classifica. Tale scelta è stata presa a causa delle dimensioni del messaggio che possono essere ben maggiori rispetto agli altri.

Nei casi invece della scelta del nickname e delle risposte al quiz il server trasmette l'esito al client per mezzo di un semplice valore booleano. In aggiunta rispetto alle specifiche richieste in caso di risposta sbagliata il server successivamente invia la risposta corretta.

Il client durante la fase di inizializzazione del gioco si fa inviare dal server i nomi dei temi e memorizza questi all'interno della sua struttura dati *temi* al fine di evitare ulteriori trasmissioni. Sempre per rispettare quest'ultimo intento, il client ricorre alla struttura dati *temi* anche per verificare quali temi siano stati già svolti dal giocatore, mostrando nel menù di scelta solo quelli ancora da svolgere.

Ogni controllo sui dati, eccetto l'univocità del nickname e la correttezza delle risposte, vengono effettuati dal client con lo scopo di evitare trasmissioni inutili al server.

Per la gestione degli errori nella comunicazione sono state definite due funzioni *gestione_recv()* e *gestione_send()* sia in *libclient* che in *libserver*. Le due funzioni gestiscono in particolare il caso in cui l'altro interlocutore della connessione chiuda il socket. Nel caso della *recv()* questa condizione si verifica quando il valore ritornato è 0 oppure -1 con *errno* = *ECONNRESET*. Per ogni *send()* invece è stato necessario passare come ultimo parametro *MSG_NOSIGNAL* al fine di non far generare al kernel il segnale *SIGPIPE* che nel caso non venga gestito esplicitamente fa terminare il processo coinvolto. In questo modo si è resa possibile l'individuazione di chiusura del socket all'interno della *send()* verificando se il valore sia -1 e se *errno* sia uguale a *ECONNRESET* o *EPIPE*. È stato necessario definire due diverse versioni delle funzioni perché mentre il client si limita a chiudere il socket, il server deve anche compiere le azioni che seguono all'uscita di un giocatore dal quiz.

Il comando *endquiz* nel client è stato implementato come una semplice *goto* che porta alla chiusura del socket prima che riparta il ciclo di gioco. All'interno del server quindi non è stato gestito specificamente: siccome non vi sono differenze da una disconnessione i controlli effettuati da *gestione_recv()* e *gestione_send()* sono già sufficienti.