Progetto di Reti Informatiche A.A. 2024/2025
Professori: Francesca Righetti, Giuseppe Anastasi
Studente: Antonio Querci

Introduzione

Gli obbiettivi principali durante la realizzazione di questo progetto sono stati l'aderenza alla traccia e l'imparare a progettare un applicazione *client-server*, cercando di sperimentare il più possibile le varie tecnologie che non avevo ancora mai usato (in particolare la programmazione distribuita). Altri obbiettivi sono stati il rendere il progetto modulare e dinamico, garantendo *information hiding* ove possibile, e il rendere il codice efficiente dal punto di vista di risorse (sia come tempo di esecuzione che come memoria utilizzata).

Codice

Il codice è stato strutturato in modo da essere modulare e dinamico. È presente un file *costanti.h* che contiene soltanto le costanti sfruttate sia da *client.c* che da *server.c*; le costanti usate da un solo modulo restano in quel modulo (quindi non si trovano in *costanti.h*), così da favorire l'*information hiding*.
Analogamente, le uniche funzioni condivise sono la *stampa_delimitatore()*, la *send_all()* e la *recv_all()*: tali funzioni sono, rispettivamente, contenute nei moduli *stampa_delimitatore.c* e *send_recv_all.c*.
La compilazione del codice avviene tramite Makefile. È presente un file *start.sh* che permette di lanciare in automatico la compilazione (tramite *make*) e di avviare il server e un client.
Per garantire la dinamicità, il numero di temi e i nomi dei temi sono letti a *runtime* dal server tramite il file *info.txt*, così che si possano aggiungere eventuali temi semplicemente modificando quel file: nella prima riga tale file contiene il numero dei temi mentre, nelle sottostanti righe, contiene i nomi dei temi.
Per ogni tema, è presente un file nominato *i.txt* dove *i* è il numero (a partire da 1) associato al tema (i numeri sono associati nell'ordine in cui i temi appaiono nel file *info.txt*). Ogni file *i.txt* contiene 5 domande e, sulla stessa riga di ogni domanda, dopo il "?" (che termina la domanda), contiene le possibili risposte valide a tale domanda, separate dal simbolo "|".

Tipologia di server

Per permettere di servire più client in contemporanea (ed è probabile che vi siano più client in contemporanea), si è scelto un server multithreaded. Al fine di risparmiare overhead e memoria, sono stati utilizzati i thread al posto dei task (con dovuta gestione della concorrenza). Il server è, dunque, un server multithreaded che contiene un flusso principale che accetta ogni nuova richiesta e le associa un thread.
La scelta dei thread pone il problema della gestione della concorrenza perché, per memorizzare le classifiche e i giocatori, ogni thread deve accedere a delle strutture dati condivise. D'altro canto, però, le strutture dati condivise si adattano bene alla realizzazione di classifiche (infatti ogni thread che gestisce un client può, in modo controllato, aggiornare queste classifiche). Ho reputato che la gestione della concorrenza fosse un problema trascurabile rispetto alla riduzione di overhead e di memoria.
Per gestire la concorrenza, sono stati usati i mutex. Il loro utilizzo è specificato nella sezione delle strutture dati.
Un server concorrente risulta adatto anche perché non vi è, in alcun modo, interazione tra client.

Dunque, visto che i client non interagiscono tra di loro, il server multithreaded permette di rispondere in parallelo alle richieste dei client, migliorando l'esperienza del servizio.

Strutture dati

Il server deve stampare frequentemente classifiche e, vista la presenza del comando *show score*, inviarle ai client. Per evitare ordinamenti frequenti - che costerebbero *O(nlog(n))* -, ho scelto di usare alberi binari di ricerca, così da avere inserimenti con costo *O(log(n())* e stampe ordinate con costo *O(n)*. In particolare, ho usato un ABR per permettere l'inserimento dei giocatori in ordine di nickname crescente, un vettore dinamico di ABR per i quiz completati (ordinato sempre per nickname crescente) e un altro vettore dinamico di ABR per le classifiche per ogni tema (ordinate in ordine di punteggio e crescente, a parità di punteggio, in ordine di nickname "decrescente" – ovvero il figlio destro di un nodo avrà un nickname alfabeticamente minore del figlio sinistro di un nodo"). I vettori sono dinamici perché il numero di quiz viene letto da file, come detto nella sezione "Codice". Per gestire la concorrenza, viene allocato un mutex per ogni ABR (quindi un mutex per l'albero dei giocatori e un mutex per ogni elemento dei vettori dinamici).

Comunicazioni su rete

Per quest'applicazione, ho scelto il protocollo TCP. Infatti, per funzionare correttamente, quest'applicazione necessita di ricevere i tutti i pacchetti e in ordine corretto.
Al fine di aumentare la flessibilità dell'applicazione, prima dell'invio di ogni messaggio, deve essere comunicata la lunghezza del messaggio al destinatario: questo è stato fatto perché la maggior parte dei messaggi ha dimensione variabile, quindi avrei sprecato memoria (soprattutto nel caso di molti client connessi) del server. L'unico caso in cui è stato definito un formato di messaggio (senza, però, una lunghezza predefinita, quindi risulta necessario comunicare al destinatario la lunghezza del messaggio stesso anche in questo caso) è per l'invio della stringa contenente le classifiche per ogni tema richiesta dal client tramite il comando *show score*. Si è preferito non definire anche in questo caso la lunghezza del messaggio visto che tale risulta fortemente variabile (per esempio, in base al numero di giocatori che stanno rispondendo alle domande e a quanti quiz ogni giocatore ha effettuato).
Quando si inviano stringhe, la marca di fine c-stringa non viene immessa nel pacchetto: ciò è stato fatto per ridurre la dimensione dei messaggi.
Vista la natura dell'applicazione, il protocolli più utilizzati sono stati di tipo text (l'unica eccezione è per i valori numeri rappresentanti lunghezze di messaggi: in quel caso si usano binary protocols). Infatti, visto che la dimensione dei messaggi è già trasmessa come binary protocol e, nella maggior parte dei casi, i messaggi non contenenti lunghezze sono stringhe, la compressione di un binary protocol risultano trascurabili rispetto ai vantaggi di un text protocol (anche nel caso di *show score*, trasmettere una stringa unica risulta vantaggioso, visto che il punteggio di ogni giocatore per ogni tema è sempre ≤ 5, quindi sta su un byte).
Per garantire il corretto invio e la corretta ricezione della quantità di byte stabilita, sono state implementate le funzioni *send_all()* e *recv_all()*, usate sia da *client.c* che da *server.c*. Tali funzioni permettono anche la gestione di errori sia da parte di *send()* che di *recv()*.
Nel caso di *send()*, per far sì che il mittente si accorga del fatto che il *socket* remoto si sia chiuso, è stata implementato un *handler* per intercettare il segnale SIGPIPE.