Progetto di Reti Informatiche

Lorenzo Marranini

A.A. 2024/2025

Documentazione

Server: È stato considerato opportuno adottare un server basato sulla funzione *select* di I/O multiplexing, al fine di gestire in modo semplice e intuitivo lo scambio di messaggi con i vari client connessi.
Questa scelta offre il vantaggio di una maggiore efficienza in presenza di un numero limitato di client, scenario che rispecchia le effettive esigenze dell'applicazione sviluppata per il progetto.
Inoltre, trattandosi di un'applicazione progettata per la gestione di un semplice quiz, che non richiede interazioni o risposte del server con tempi estremamente rapidi, la gestione di tutti i client da parte di un unico processo non compromette in alcun modo l'esperienza lato client.

È stata adottata la modalità di scambio dati in formato testuale, considerando che la maggior parte dei messaggi sono composti da testo. Non è stato definito un formato di messaggio predefinito, poiché i messaggi variano significativamente per forma e lunghezza. Di conseguenza, lo scambio di messaggi è stato implementato in modo tale che il destinatario venga preliminarmente informato sul numero di byte del messaggio da ricevere.

La struttura dati *Client* è ciò che rappresenta e mantiene le informazioni di un client che ha stabilito una connessione, questa contiene le informazioni essenziali come l'identificativo del *socket*, il *nickname*, il tipo di quiz scelto, il punteggio del quiz che in quel momento è in esecuzione, la domanda a cui è fermo il client e un puntatore ad il prossimo Client poiché l'insieme di Client connessi sono stati gestiti con una lista che ha come puntatore di testa *head_clients*; infine contiene due puntatori di tipo *Client_points* che puntano ai nodi corrispondenti presenti nelle due liste che rappresentano le due classifiche (una per ciascun tema).

La struttura dati *Client_points* è stata creata per mantenere le informazioni essenziali solo per gestire i punteggi e le classifiche in tempo reale sia dei quiz attivi, che di quelli terminati: contiene infatti solo il nickname, il punteggio, una variabile *terminated_quiz* che indica se questo è terminato, il puntatore al nodo precedente e infine il puntatore al nodo successivo.
Le due liste sono sempre tenute ordinate secondo lo score del quiz in tempo reale di ciascun client e hanno come puntatore di testa *head_classifica1* e *head_classifica2.*

Queste due nuove liste potrebbero sembrare una ridondanza di informazioni già contenute nella lista dei client, ma sono state create per 2 motivi:
1) Non è possibile avere la lista dei client ordinata secondo un unico *score* poiché i quiz sono due e gli score potrebbero essere due nel caso in cui un client termini un quiz e poi inizi l'altro;
2) Utilizzando liste separate, queste contengono il numero minimo ed essenziale di informazioni che permettere di stamparle a video in modo molto veloce e favorisce anche l'invio delle informazioni al client quando questo richiede di vedere le classifiche con il comando *show score.*

Questa scelta ha portato alla creazione di molte funzioni per gestire la creazione, eliminazione e riordinamento dei nodi delle liste che rappresentano le due classifiche, nonostante l'overhead introdotto questo permette di non sprecare spazio inutilmente sul server e migliorare l'efficienza di altre funzioni che vengono richiamate con maggiore frequenza come la funzione *outputserver* che mostra a video del server i vari partecipanti attivi e i loro punteggi; mentre le funzioni di gestione delle liste sono chiamate con minore frequenza.
Per cercare di rendere il minore possibile l'overhead sono presenti i due campi node1 e node2 nella struct *Client* così da avere un riferimento valido in ogni istante (se il client non ha ancora iniziato un quiz di un tema node1 o node2 saranno NULL); e oltre a questi nella struct *Client_points* il campo *prec* che permette di riordinare la classifica quando uno score viene aumentato in modo veloce senza ripercorrere l'intera lista inutilmente.

Le domande e le risposte sono contenute nei file *domande.txt*, e r*isposte.txt* e ciascuna è separata dall'altra dal ritorno carrello; le domande sono numerate direttamente nel file di testo e sono scritte in modo sequenziale; quindi, le prime *MAX_QUESTIONS* (macro definita all'inizio del codice) sono quelle del primo tema, e le successive del secondo.
Queste vengono caricare all'interno di vettori di caratteri all'inizio del main, grazie alla funzione *load_questions_and_answers.*
I file con le domande e risposte devono essere nella stella directory in cui è presente il file del server.

Una volta creato il server del socket ed essere entrati nel ciclo while infinito, grazie alla select si dividono i messaggi ricevuti dai client in 2 tipi: messaggi da nuovi client e messaggi da client già connessi, gestite rispettivamente da h*andle_new_connection* e *handle_client_message*:
Nel primo caso, se non si è raggiunto il numero max di clients già connessi, si invia il messaggio per richiedere il nickname.
Nel secondo caso grazie alla variabile *clientstate* si interpreta il messaggio ricevuto dal client in modo opportuno: può essere il nickname, il tipo di quiz che si vuole iniziare, una risposta ad una domanda, oppure un comando *endquiz*, *show score* o *restart*; questa funzione è vantaggiosa poiché modulare in base allo stato ma è molto complessa e in diverse parti ripete delle parti di codice più volte.

Quando un client si registra con il nickname viene istanziato un nuovo nodo del tipo struct *client_points* che mantiene le sue informazioni nella classifica del quiz attivo in quel momento, è stato deciso che nel caso in cui venga eseguito due volte lo stesso quiz, le informazioni sul precedente tentativo vengono eliminate.

Il server è stato implementato in modo che mandi sempre un messaggio al client e dopo si aspetti sempre una risposta che sia significativa, questo per favorire la semplicità del codice del client e per utilizzare un protocollo semplice in cui il server si aspetta sempre un messaggio in risposta all'ultimo messaggio da lui inviato.

Per questo motivo nella funzione *showscore* è passato come argomento il set di domande del quiz attivo dal client in quel momento, poiché al messaggio contenente le informazioni sui punteggi e sui partecipanti deve essere concatenata la domanda a cui si era eventualmente interrotto il quiz, così che la risposta successiva del client sia interpretata come una risposta a quest'ultima.

La funzione *check_answer* ha il vantaggio di dare come corretta una risposta in modo case-insensitive, ma non gestisce la presenza di articoli davanti alla risposta o di sinonimi che potrebbero essere comunque validi di uso comune nella lingua italiana.

La funzione *handle_sigint* gestisce la chiusura ordinata del server ma non garantisce il salvataggio dello stato corrente prima della terminazione, questo non era richiesto nelle specifiche e di conseguenza i dati dei client al termine del server vanno persi.

In conclusione, il server presenta una gestione dinamica delle strutture dati dei client, una modularità nelle funzioni che gestiscono i messaggi ricevuti e una semplicità dovuta alla select, però presenta overhead nelle operazioni che gestiscono le liste, è priva di gestione degli errori di rete e file (presenta solo la notifica di questi ultimi per la maggior parte delle operazioni) ed è difficilmente estendibile in termini di quiz e funzionalità.
Nonostante ciò risponde perfettamente alle esigenze e alle capacità necessarie presentate nei requisiti di progetto.

Client: il client è molto più semplice e inizialmente mostra solo a video la prima domanda che è quella che determinerà la richiesta di connessione al server o la terminazione del programma.

La funzione *run_client* permette di essere richiamata ogni volta che il client risponde con endquiz e poi però ricomincia il quiz da capo.

Una volta stabilita la connessione si entra nel ciclo while infinito e si effettua sempre lo stesso meccanismo: ricezione di un messaggio e invio di un messaggio di risposta.

Il client presenta quindi dei vantaggi come la semplicità, la gestione chiara degli errori e l'efficienza elevata nel caso di un'applicazione ridotta come questa.
Tra gli svantaggi non è stato gestito un eventuale time-out o inattività del server, la gestione di interazione multipla a più server in contemporanea, e non è presente un tipo di configurazione dinamica.

Nell'utilizzo di questa applicazione di rete, si comporta in modo efficace rispettando i requisiti di progetto e rispondendo in modo corretto ad eventuali errori che si potrebbero verificare dal lato server o lato client.