Documentazione progetto Reti Informatiche

Scelta del Protocollo

Il protocollo scelto per la comunicazione tra client e server è **TCP**. Questa scelta garantisce la corretta trasmissione dei dati e l'ordinamento dei messaggi, fondamentale per la gestione delle domande e delle risposte del quiz. Sebbene TCP possa introdurre un overhead maggiore rispetto a UDP, nel contesto del progetto l'affidabilità è più importante della reattività.

Per la trasmissione dei dati è stato adottato un protocollo **testuale**, strutturato in messaggi prefissati o 'tokenizzati', che consente un'implementazione semplice e facilmente estendibile.
Ad esempio, possiamo trovare:

- *START_Partecipo*: il client vuole partecipare al quiz.
- *NCK_nickname*: il client invia il proprio nickname.
- *TEMA_Tech* o *TEMA_Gen*: il client seleziona il tema delle domande.
- *ANSWER_risposta*: il client invia una risposta.

Questa scelta facilita il debug e rende l'applicazione intuitiva. Tuttavia, un protocollo testuale introduce un leggero overhead rispetto a un protocollo binario.

I/O Multiplexing

Il server è stato progettato utilizzando **I/O Multiplexing** tramite la funzione select().
Questa scelta permette di gestire più client contemporaneamente senza la necessità di thread o processi multipli, mantenendo il server efficiente e scalabile per un numero moderato di connessioni.

Il server mantiene un **set principale** (fd_master) per tracciare i socket attivi e un **set di lettura** (fd_read) per monitorare gli eventi in arrivo. Il socket principale (listener) è responsabile dell'accettazione delle nuove connessioni, mentre i socket dei client vengono gestiti in base ai messaggi ricevuti con i loro rispettivi token.
Tramite l'I/O Multiplexing possiamo evitare il blocco di attesa da parte del server su un singolo client, consentendo al server di essere sempre disponibile, e riduciamo l'overhead rispetto ad un server multi-threaded.
Sicuramente questa scelta non agevola un numero molto elevato di connessioni poiché la funzione select() oltre ad avere un limite sul numero massimo di socket che può gestire, ad ogni chiamata deve copiare l'intero set di file descriptor, portando un overhead significativo.

Strutture Dati

La gestione dei partecipanti e dei punteggi è affidata a una struttura **LeaderboardEntry**, che contiene:

- **Nickname**: identificativo del client.
- **Socket associato**: per identificare il partecipante attivo.
- **Punteggi per tema**: score_tech e score_general.
- **Progresso nel quiz**: current_question e theme_index.
- **Completamento del tema**: flag theme_tech_completed e theme_gen_completed.

Questa struttura è dinamicamente aggiornata durante l'interazione del quiz, con funzioni per l'inserimento, l'ordinamento e la rimozione dei partecipanti.

Gestione del Quiz

Le domande sono caricate da file di testo (tech_questions.txt e general_questions.txt) all'avvio del server. Durante lo svolgimento del quiz:

1. Il server rende riconoscibile il client nel quiz chiedendogli un nickname, univoco.
2. Effettuata la scelta del 'Tema', il server inizierà inviando la prima domanda al client, aggiorna lo stato del quiz e si mette in attesa della risposta.
3. La risposta del client viene valutata e il punteggio aggiornato.
4. Il client riceve l'esito della risposta e un menù dei comandi (*next*, *showscore* e *endquiz*) per procedere eventualmente alla prossima domanda.
5. Al termine del quiz il server aggiornerà la struttura del partecipante e inoltrerà il punteggio effettuato dal client nel quiz, riproponendo a quest'ultimo di giocare senza dover chiudere la connessione o cambiare nickname.

Nel progetto, lo scambio di messaggi tra client e server è stato progettato utilizzando le funzioni personalizzate **doppia_send()** e **doppia_recv()**, che seguono un protocollo a due fasi:

- Prima di ricevere o inviare il messaggio vero e proprio, viene inviata (o ricevuta) la lunghezza, convertita nel modo corretto, del messaggio sotto forma di un intero.

- Una volta nota la lunghezza, viene trasferito il messaggio vero e proprio, utilizzando esattamente il numero di byte necessari.

Quindi, non è necessario leggere o inviare sempre l'intero buffer di dimensione fissa (es. *BUFFER_SIZE*). Invece, vengono trasmessi solo i byte necessari, risparmiando banda e riducendo il carico di lavoro.
Leggendo esattamente il numero di byte necessario, si evitano situazioni in cui il buffer contiene dati residui di messaggi precedenti o viene letto oltre i limiti.

Criticità e Miglioramenti

Ad ogni iterazione, select() controlla tutti i socket monitorati, anche quelli inattivi. Questo introduce un carico aggiuntivo che, sebbene non problematico per pochi client, potrebbe diventare significativo con un numero maggiore. Ad esempio, se il server non avesse un limite di client da poter gestire, ed uno solo di loro invia messaggi, il server scorrerà tutti gli n socket per verificare chi ha i dati disponibili.

Al momento, il server rileva correttamente la disconnessione di un client, eliminandolo dalla leaderboard azzerando quindi i suoi punteggi. Tuttavia, i client attivi non vengono informati della disconnessione degli altri partecipanti.
Quando un client si disconnette, il server potrebbe inviare un messaggio di avviso ai partecipanti del quiz del client disconnesso.

Attualmente, la leaderboard viene generata e inviata su richiesta. In uno scenario con molti client, questa operazione potrebbe diventare onerosa. Mantenere una leaderboard già ordinata lato server e aggiornarla solo quando necessario potrebbe essere un buon miglioramento (ad esempio, quando un client termina un quiz o si disconnette).