Documentazione per il progetto di Reti Informatiche

(A.A. 2023/2024)

Francesco De Lucchini [635502]

Tutti i messaggi non banali** tra client e server vengono scambiati attraverso il seguente **protocollo**:

1. Primo messaggio (formato **binary**): contiene la dimensione, big endian su 16 bit, del secondo messaggio
2. Secondo messaggio (formato **text**):
    a. Primo byte: contiene un AZIONE (tipo enumerazione, troncato ad 8 bit)
    b. Secondo byte in poi: contiene una lista (opzionale) di parametri, ognuno delimitato da un carattere speciale (configurabile, di default la tilde), l'ultimo parametro è delimitato da '\0' (sapendo la dimensione non è strettamente necessario, ma rende le operazioni di decodifica e debug più semplici)

Per fare un esempio il secondo messaggio può essere "8cavo~computer\0", che corrisponde al comando: *USE cavo computer*. Tutti i dettagli si trovano in "protocol.h".

***Un **messaggio banale** è un messaggio che comporta il solo scambio di un numero intero, ad esempio, la risposta del server ad un tentativo di login viene semplicemente codificata attraverso il tipo enumerazione RESPONSE.*

Questa scelta permette di utilizzare un unico protocollo per ogni tipo di comunicazione, inviando i soli byte strettamente necessari. Tuttavia, rispetto ad un protocollo a lunghezza fissa, richiede due messaggi anziché uno, ed una codifica/decodifica leggermente più complessa. Bisogna infine prestare particolare attenzione ad evitare *memory leak*: essendo i **campi flessibili**, questi vengono allocati a runtime, e devono dunque essere deallocati al termine del loro utilizzo.

Tutti gli input dell'utente e tutte le richieste provenienti dalla rete sono accuratamente controllate in modo da evitare buffer overflow. Tutti gli errori su send e recv sono gestiti. In generale, si è cercato di programmare in modo **memory-safe**, con alcune funzioni della *stdlib* ridefinite a questo scopo (*ssstrlen* e *fgetsnn*, definite in "mystdlib.h").

Il server gestisce un numero (potenzialmente) illimitato di client, mantenendo con ognuno un'unica connessione TCP aperta per tutta la partita. Il protocollo scelto è appunto **TCP** perché è fondamentale che i dati arrivino corretti e nell'ordine in cui sono stati inviati. Nel mondo reale, affidarsi ad una singola connessione TCP nell'arco di tutta la partita potrebbe non essere una scelta adatta, in quanto questa potrebbe essere terminata prima del previsto da router/firewall intermedi.

È stato assunto che le operazioni di send e recv inviino e ricevano l'intero messaggio ad ogni chiamata. In una situazione reale sarebbe stato opportuno inserirle in un ciclo while per assicurarsi dell'invio e della ricezione completa dei messaggi.

I socket utilizzati sono di tipo **bloccante,** perché il server deve non deve effettuare operazioni asincrone, e le richieste sono gestite tramite la tecnica dell'**io-multiplexing** (primitiva select). In ottica di scalabilità è sicuramente più opportuno optare per un approccio, sempre io-multiplexing, ma distribuito su più processi e/o thread.

Il server non fa mai attesa attiva di una risposta del client, neanche quando gli viene posto un enigma, vengono infatti salvate in sessione informazioni che, quando arriva la risposta, permettono di distinguere la domanda alla quale il client deve rispondere. Una **sessione** ("session.h") è una struttura dati che mantiene le informazioni di un client (username, stanza attuale, oggetti sbloccati, utilizzati, token, …). Le sessioni sono organizzate in una lista, vengono allocate quando il client si connette e deallocate quando si disconnette.

Il **database** degli utenti è astratto dall'interfaccia "database.h", che fornisce due classiche primitive: *db_read*() e *db_write*(). L'implementazione sottostante è una lista di record, dove ognuno contiene i campi username e password, di dimensione massima prefissata.

Le **escape room** hanno tutte la stessa struttura ("rooms.h"), cosa che permette di definirne facilmente ulteriori oltre a quella proposta, e vengono caricate in memoria del server al suo avvio. Ogni volta che il client esegue un comando gli vengono inviate a runtime le informazioni richieste (ad esempio, se esegue "look", la stringa "Ti trovi in una stanza buia…" gli viene inviata dal server). In ottica di scalabilità questo potrebbe generare molto traffico ed essere dunque un bottleneck, si potrebbero perciò memorizzare nei client almeno tutte le possibili risposte a "look" (le più ingombranti), ed usare il server solamente per validare i comandi.

Funzionalità a piacere: le stanze sono single-player, tuttavia, se un secondo client prova a connettersi ad una stanza occupata, gli viene posta una domanda inerente al tema della stanza; se la indovina penalizza (toglie tempo) il giocatore al suo interno, se la sbaglia lo avvantaggia (aggiunge tempo). Per questo motivo non esiste l'eseguibile "other", ogni client è uguale.

Nota: è stato aggiunto il **comando "drop"** per rilasciare un oggetto.