Corso di Laurea in Informatica

PROGETTO PER ESAME DEL CORSO LABORATORIO II

”Paroliere” videogioco scritto in C, multi-threaded, CLI, POSIX, client/server

Professori:
Patrizio Dazi, Luca Ferrucci

Candidato:
Giulio Nisi

ANNO ACCADEMICO 2023/2024

Indice

Note

Si consiglia la lettura del codice in caso di necessitá, in quanto é stato molto commentato, forse anche pedantemente (piú del necessario) e quindi potrebbe risultare piú chiaro di tale relazione riassuntiva. Il codice é stato scritto in inglese per essere compreso potenzialmente da un pubblico piú ampio e pubblicato su GitHub.

> ”Scrivete programmi che facciano una cosa e che la facciano bene. Scrivete programmi che funzionino insieme. Scrivete programmi che gestiscano flussi di testo, perché quella é un’interfaccia universale.”
>
> *Peter H. Salus, A Quarter Century of Unix, Unix philosophy.*

1. Strutture dati ed algoritmi

1.1 Server

1.1.1 Lista giocatori

La struttura dati principale usata per la gestione dei clients/giocatori, é una lista concatenata di elementi con una struttura e puntatore al prossimo elemento della lista. É stata scelta per:

- Possibilitá di gestire potenzialmente infiniti giocatori, risorse computazionali permettendo.

- Allocazione dinamica on demand che permette di sprecare poca memoria (non preallocandola) quando i giocatori si scollegano (la memoria viene liberata), o sono assenti.

- Il vantaggio che tutte le operazioni riguardanti un solo determinato client possono essere svolte dal relativo thread attraverso il puntatore alla struttura rappresentante suddetto giocatore e qui si hanno tutte le informazioni che potrebbero servire.

Lo svantaggio sostanziale é che talvolta dobbiamo scorrere tutta la lista sincronizzandoci opportunamente con particolare attenzione, perché, in caso contrario, a differenza di un array per esempio, potrebbero verificarsi problemi molto piú gravi in certe fasi, come nell'aggiunta o rimozioni di elementi. Per far ció si utilizza (ma non é l'unico) listmutex.
Tale lista é condivisa tra clients registrati e non.

1.1.2 Matrice di gioco

La matrice di gioco, avendo dimensione predeterminata alla compilazione e costante é allocata staticamente globalmente come matrice di caratteri (char[][]). Il 'Qu' viene salvato come 'Q', ma gestito opportunamente.

1.1.3 Parole e dizionario

Le parole del file dizionario sono memorizzate in un array (words) di stringhe (char**) allocato dinamicamente, sia l'array, che le stringhe stesse. Le parole sono caricate tutte in memoria, leggendo il file, solo una volta inizialmente, limitando l'impatto delle costose operazioni di I/O con il disco. Vi é poi una copia di words (words_valid) allocata in memoria, dinamicamente, peró, NON riallocando le stringhe ed utilizzando inutilmente altra memoria, ma semplicemente copiando i puntatori da words. Ad ogni inizio nuovo gioco, contestualmente al cambio della matrice di gioco, i puntatori alle stringhe di words_valid vengono aggiornati, quelli delle parole presenti nella nuova matrice, rimangono invariati (copiati da words), mentre quelli delle parole NON presenti, vengono incrementati fino a raggiungere il terminatore (\0) di stringa. Vi é poi un'altra copia di words_valid (words_validated), allocata identicamente, per ogni giocatore registrato, il cui puntatore si trova nella struttura rappresentate tale giocatore. Quando una parola valida (presente in words_valid E in words_validated) viene sottomessa da un utente registrato, per la PRIMA volta, words_validated viene aggiornato, incrementandone il puntatore fino al terminatore, cosícché ad nuova sua sottomissione della medesima parola, essa possa essere rifiutata senza attribuirne nuovamente il punteggio al player.
Riassumendo, le parole del file dizionario vengono tutte cercate ad ogni cambio di matrice di gioco (inizio nuovo gioco) (ma NON lette dal file, vengono caricate tutte solo la prima volta in words) nella matrice stessa per aggiornare words_validated, del quale ogni giocatore (registrato) avrá una copia personale per tracciare quelle giá sottomesse.

1.1.4 Algoritmo di ricerca

Per gestire la ricerca delle parole proposte si é scelto di utilizzare il seguente algoritmo. Per ogni parola viene visitata tutta la matrice e per ogni suo carattere (della matrice) invocata la funzione searchWordInMatrix(), la quale parte dal carattere della matrice specificato (con indici i di riga e j di colonna della corrente iterazione) e ricorsivamente controlla che i caratteri adiacenti [(i, j+1),(i+1, j),(i,j-1),(i-1,j)] nella matrice siano corrispondenti al

prossimo carattere di parola cercato. Se non vi é corrispondenza, o i o j sono fuori dal range di grandezza della matrice, si ritorna un fallimento, altrimenti, se il prossimo carattere di parola voluto é identico al terminatore di stringa, allora significa che precedentemente abbiamo trovato tutte le lettere disposte adiacentemente e quindi ritorniamo un successo. Bisogna prestare attenzione a "marcare" i caratteri giá visitanti per evitare di poter utilizzare piú volte lo stesso carattere della matrice nella composizione di una parola o di perdersi in loop infiniti. Per quanto riguarda la complessitá? Vediamo un esempio. Per semplicitá assumiamo di avere M una matrice N x N (utilizzando sempre matrici quadrate), di avere K parole sul file dizionario, e di avere una parola W, da cercare, in input, ossia verificarne la presenza nella lista delle parole E nella matrice. Con la predetta soluzione, dopo aver letto dal file le parole ed averle inserite in words, si dovrá riempire words_valid (solo una volta ad ogni inizio gioco), scorreremo quindi words in K passi, ad ogni passo iteriamo su M effettuando $N \cdot N$ passi, invocando ad ognuno, searchWordInMatrix(), la quale per trovare la parola al massimo effettuerá proprio altri $N \cdot N$ passi che rappresentano la parola di lunghezza massima che si puó trovare nella matrice, vediamo a seguire un esempio con N = 5 dove con la numerazione é riportato solamente uno dei possibili percorsi completi che searchWordInMatrix() potrá intraprendere per trovare

la parola di lunghezza massima 25: $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 10 & 9 & 8 & 7 & 6 \\ 11 & 12 & 13 & 14 & 15 \\ 20 & 19 & 18 & 17 & 16 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$.

Quindi ricapitolando $K \cdot N \cdot N \cdot N \cdot N = N^4K$ la pima volta, a seguire, ogni parola cercata da un giocatore comporterá solamente la ricerca in words_valid (ed il controllo in words_validated, ma questo si fará ad accesso diretto se trovata, dato che gli array sono allineati) con costo K. In conclusione credo sia una discreta implementazione, che sfrutta la potenza dell'aritmetica dei puntatori, le stringhe sono effettivamente in memoria solo una volta e a fronte di un costo iniziale piú elevato permette di rispondere piú velocemente a seguire ai giocatori.

1.1.5 Coda di fine gioco

Per la struttura del progetto vista, la coda richiesta dalle specifiche, non sarebbe stata necessaria, anzi é risultata un'inutile complicazione. Sarebbe stato sufficiente sbloccare i threads dei clients e segnalarli di inviare il messaggio di fine gioco con la scoreboard al client gestito. Ogni thread cosí, sfruttando la peculiaretá sopra spiegata, avrebbe potuto inviare il messaggio utilizzando il puntatore alla struttura del giocatore in possesso che fornisce giá tutte le informazioni necessarie. Comunque per aderenza alle richieste, la coda é stata utilizzata. Essa é stata implementata come lista concatenata, ogni elemento é una struttura contenente un puntatore al corrispettivo client, un puntatore ad un messaggio (con struttura come richiesta) contenente nel campo data, il nome del client (se registrato, un placeholder altrimenti) ed il punteggio ottenuto nel gioco separati con una virgola, ed infine un puntatore all'elemento successivo. Per la sincronizzazione viene utilizzato (ma non é l'unico) queuemutex.

1.2 Client

Nel client vengono utilizzate due liste allocate dinamicamente. La prima di una struttura contenente stringhe allocate dinamicamente, dove ogni stringa ha una lunghezza massima BUFFER_SIZE. La seconda di una struttura contenente messaggi che rappresentano le risposte ricevute dal server. Per sapere il peculiare motivo di queste fare riferimento alle pagine seguenti. Nessun particolare algoritmo é utilizzato.

2. Struttura codice

Il codice si trova tutto nella carte Src (soruces).
Le dipendenze sono le seguenti:

boggle_server (paroliere_srv) eseguibile dipende da boggle_server.c, server.c, common.c..
boggle_client (paroliere_cl) eseguibile dipende da boggle_client.c client.c common.c..

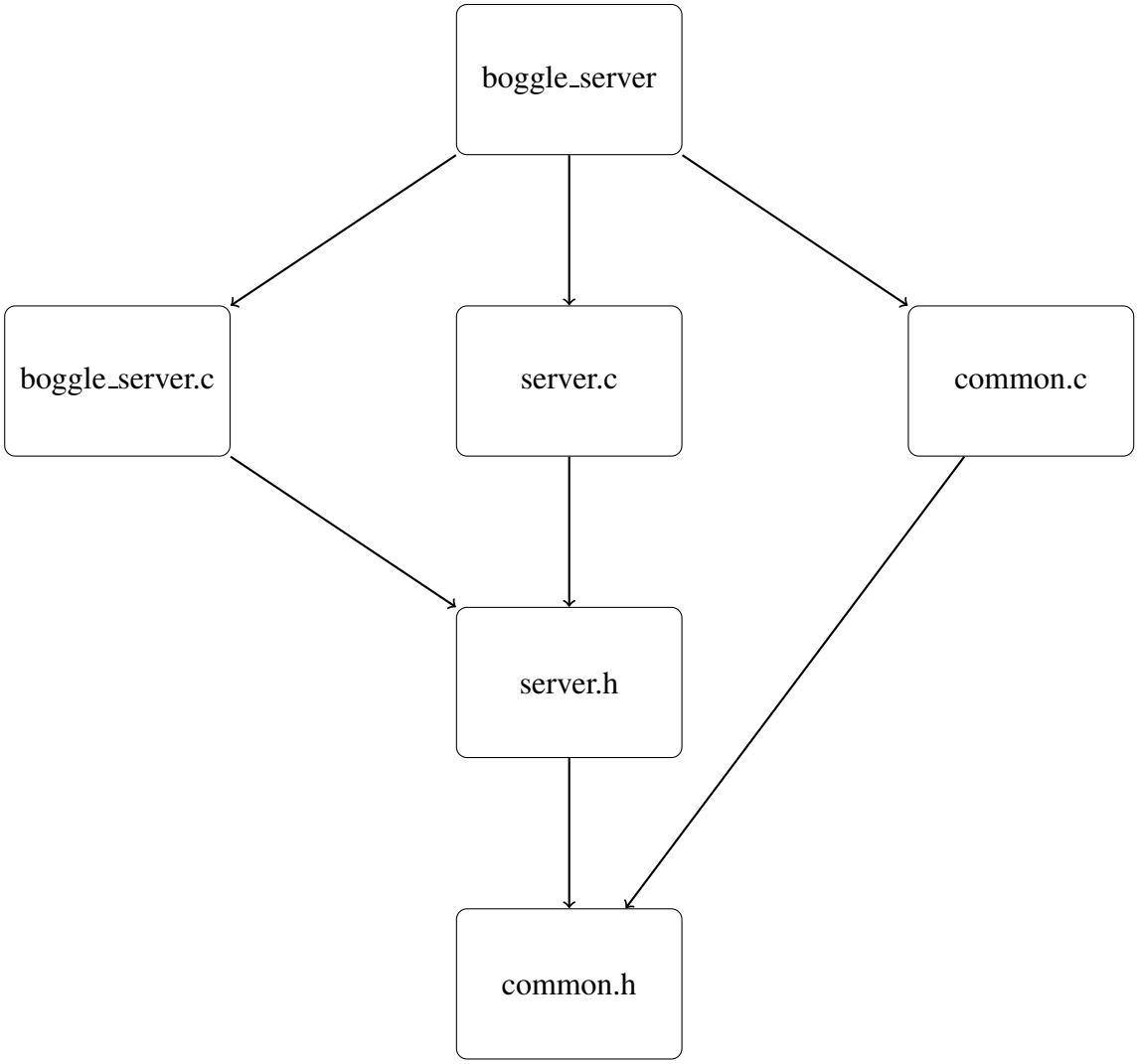
boggle_server.c, server.c dipendono da server.h..
boggle_client.c, client.c dipendono da client.h..

server.h, client.h dipendono da common.h..

common.c dipende da common.h..

Si noti che common.h e common.c includono dichiarazioni ed implementazioni comuni a server e client.

A seguire un diagramma delle dipendenze per il server. Quello del client é analogo.


boggle_server
boggle_server.c
server.c
common.c
server.h
common.h

3. Struttura programmi

Struttura programmi...

4. Struttura tests

Tests...

5. Compilazione ed esecuzione

Compilazione...