IL PAROLIERE ONLINE

Progetto Finale del corso di Laboratorio 2A – a.a. 2023/2024
Professori Patrizio Dazzi e Luca Ferrucci

Progetto e Relazione a cura di Vicarelli Tommaso – Matricola 638912

DESCRIZIONE GENERALE
Il gioco consiste nel comporre parole utilizzando le lettere posizionate su di una matrice quadrata di 16 lettere (4x4). Ogni parola deve essere formata da almeno 4 caratteri alfabetici e deve essere composta utilizzando le lettere della matrice una sola volta. Le caselle utilizzate devono essere contigue e composte in senso orizzontale e verticale. Non è possibile passare sopra una casella più di una volta. Se la parola è conforme alla matrice, appartiene al dizionario italiano e non è stata già proposta, allora vengono attribuiti tanti punti quanto la lunghezza della parola.

STRUTTURAZIONE E ORGANIZZAZIONE DEI FILE

Per una maggiore comprensione e strutturazione del progetto, i vari file sono stati catalogati in cartelle distinte in base al loro tipo:

- Cartella “Principale”: contiene tutte le sottocartelle, questa relazione in formato PDF, i file di testo contenenti rispettivamente il dizionario e le matrici, e il Makefile per la compilazione e l’esecuzione del progetto
- Sottocartella “Header”: contiene tutti i file libreria del progetto, ossia i file .h
- Sottocartella “Sorgente:” contiene tutti i file sorgente del progetto, ossia i file .c

Ogni file sorgente ha il proprio header corrispondente, fatta eccezione per server e client che contengono il corpo del codice.
Nel file *serverfunction.c* sono contenute tutte le implementazioni delle funzioni usate dentro il file *server.c*, dalla gestione delle liste, al TRIE, fino all’inizializzazione della matrice.
Nel file *serverfunction.h* sono contenute tutte le dichiarazioni delle strutture utilizzate in questo progetto.
Nel file *comunication.c* sono contenute le due funzioni *sender* e *receiver* che si occupano di caricare sul file descriptor i dati di scambio tra client-server e server-client, infatti questo file è sfruttato sia dal client che dal server.

STRUTTURAZIONE LOGICA DEL SERVER

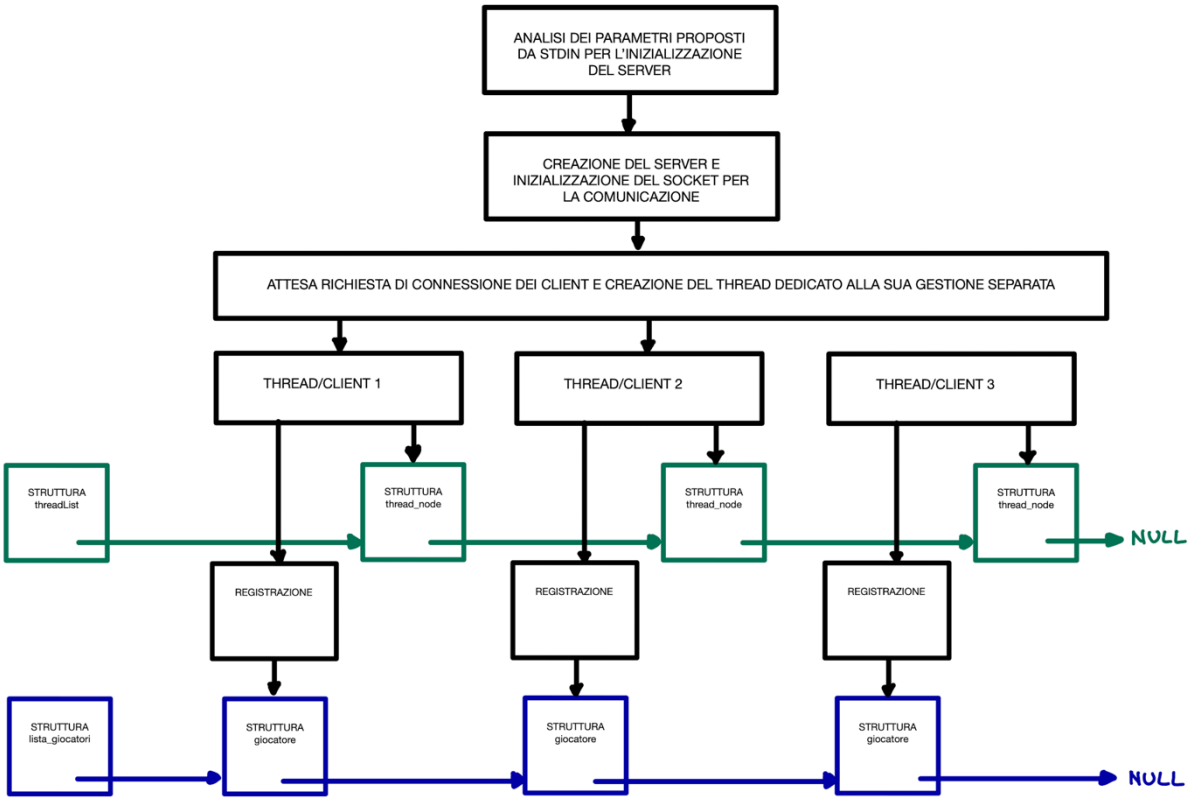
Il file *server.c* si occupa di gestire tutte le meccaniche di avvio del server, accettazione dei giocatori e sviluppo del gioco, accompagnato dai file sorgente *serverfunction.c* e *comunication.c*.
Il server è gestito tramite un approccio mutithread.

Processo Main: all’avvio tramite eseguibile, preleva i parametri dallo STDIN. Per prima cosa controlla i parametri obbligatori, cioè *nome_server* e *porta*, perché senza di loro non si ha modo di dare il via al server.
Successivamente si passa all’accettazione, se presenti, dei parametri opzionali, controllando tramite le long option il tipo di parametro fornito e assegnandolo alla corretta variabile globale che servirà per dare delle opzioni avanzate rispetto alle meccaniche di gioco di default.

A questo punto avviene il caricamento del dizionario che per essere efficiente in lettura viene caricato in una particolare struttura chiamata Trie. Questa struttura consiste in un albero utilizzato per memorizzare un insieme di stringhe. Ogni nodo rappresenta un prefisso comune delle stringhe memorizzate, e i cammini dall'albero radice alle foglie rappresentano le stringhe complete.

Una volta che è tutto pronto si chiama la funzione *server()* che si occuperà di inizializzare il socket di comunicazione e di mettersi in attesa di nuove connessioni.
Viene inoltre creato il thread *game* che si occuperà della gestione del tempo e delle meccaniche di gioco come la preparazione del round e l'invocazione dello *scorer.*

Quando viene accettata una nuova connessione, viene creato il thread che si occuperà di gestire quel determinato client tramite la funzione *gestore_thread()* a cui viene passato il fd.
Verranno aggiunti i relativi fd e tid alla struttura *threadList* che memorizza le informazioni di tutti i client che hanno richiesto una connessione (registrati e non).



Thread Gestore Client: all'avvio del thread viene memorizzato subito il file descriptor passato come argomento alla funzione *gestore_thread().*
Il client entra quindi in un loop infinito dal quale si esce solo con l'avvenuta registrazione del giocatore. Durante questa fase possono essere proposti i comandi "*registra_utente username*", "*aiuto*" (gestito dal client senza che invii nulla al server) e "*fine*".
Si può uscire da questa fase solo quando il client inserisce un username valido, cioè che è univoco, ha lunghezza minore di undici caratteri ed è formato solo da caratteri minuscoli e alfanumerici.
Se viola una di queste condizioni allora il client deve ritentare la registrazione.

Per fare chiarezza, il server accetta infinite connessioni e quindi crea infiniti thread, ma solo 32 possono registrarsi e quindi giocare contemporaneamente. Gli altri possono rimanere in attesa aspettando che qualcuno si disconnetta.
Una volta convalidato l'username, viene creato un nuovo giocatore nella struttura *lista_giocatori*, la quale contiene tutti i giocatori registrati correttamente e abilitati a giocare.

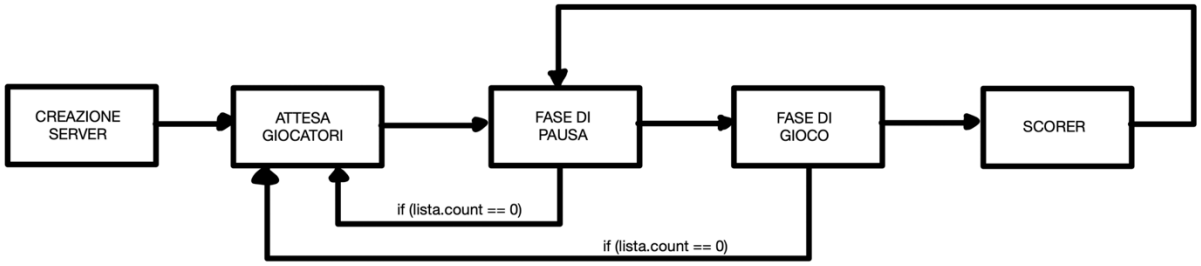
Viene poi registrato il segnale *SIGUSR1* che comunica al thread che lo riceve di fare determinate azioni: il thread giocatore deve prendere il proprio username e il punteggio dalla propria struttura giocatore, memorizzarlo localmente e aggiornare il punteggio della struttura a 0 così da prepararlo per il prossimo round. Successivamente fa una push nella *risList* dei propri username e punteggio memorizzati localmente, segnalando poi allo *scorer* tramite una condition variables che c'è qualcosa da prelevare dalla lista dei risultati.

Viene inizializzata la lista *paroleTrovate* che contiene tutte le parole trovate in un round. Questa lista verrà inizializzata all'inizio di ogni gioco.

In base alla fase di gioco in cui ci troviamo viene mandato al giocatore la matrice e il tempo di gioco rimanente, se ci troviamo in un round, oppure il tempo residuo all'inizio della prossima partita, se ci troviamo in pausa.
Inizia così il vero e proprio loop di ricezione dei comandi di gioco. In base alla fase in cui ci troviamo, se di gioco o di pausa, il server può accettare diversi comandi:

- *matrice*, questo comando restituisce la matrice attuale e il tempo residuo del round se ci troviamo nella fase di gioco, altrimenti restituisce il tempo rimanente all'inizio della prossima partita.
- *p parola*, anche questo comando varia in base alla fase in cui ci troviamo. Ovviamente se siamo in pausa non è possibile proporre delle parole, quindi verrà ritornato al client un messaggio di errore. Se invece ci troviamo nella fase di gioco, allora verrà controllata la validità della parola proposta, cioè se contiene solo caratteri alfabetici minuscoli, se la parola è già stata proposta (quindi se appartiene alla lista *paroleTrovate*), se è presente nella matrice e se è presente nel dizionario. Una volta superati tutti questi controlli, la parola viene considerata valida, viene quindi aggiunta alla lista delle parole trovate, viene aumentato il punteggio sia locale che nella struttura giocatore, ma viene restituito anche al client. In tutti gli altri casi viene inviato al client un messaggio di errore in base alla non conformità della parola. L'unico caso in cui viene ritornato un messaggio diverso da quello di errore è quando viene proposta una parola già presente nella lista *paroleTrovate*: in quel caso viene inviato al client punteggio 0.
- *registra_utente username*, questo comando non è abilitato visto che il giocatore si è già registrato correttamente, quindi viene restituito al client un messaggio di errore.
- *classifica*, se ci troviamo in pausa e la variabile *classificaBool* (serve per capire se la classifica è stata prodotta oppure no) è 1, allora viene inviata al client la classifica in formato CSV e il tempo rimanente all'inizio della prossima partita, altrimenti viene inviato un messaggio di errore.
- *fine o SIGINT* da client, questo comando notifica al server che deve provvedere ad eliminare il giocatore da tutte le strutture in cui è stato memorizzato, quindi *lista_giocatori* e *threadList*, e a chiudere il relativo thread che lo gestiva, rendendo così disponibile lo slot per un nuovo giocatore che si vuole registrare.

Thread game: all'avvio del thread viene istanziata una variabile che ci segnala se il round è stato preparato o no, così nel caso di round pronto (e quindi non utilizzato) non andiamo a ripreparlo e a buttare, se esiste, una riga del file delle matrici.
Inizia così il loop infinito del gioco così strutturato:



Appena viene aperto il server, il game si mette in attesa di nuovi giocatori registrati. Appena se ne registra uno inizia la pausa, settando l'alarm con la durata della pausa di 60sec e memorizza il tempo di inizio. Successivamente controlla se il round è stato preparato o no, in caso negativo lo prepara prendendo la riga dal file *matrix*, se presente, oppure generando matrici casuali se il file non è presente o abbiamo finito le righe disponibili. Sfrutto un while per simulare l'attesa della pausa, il quale verrà interrotto dal cambio della variabile *pausa_gioco* da 1 in 0, effettuato dall'handler della *SIGALARM* allo scadere del tempo di pausa.
Se alla fine della pausa non ci sono giocatori registrati, si ripete il ciclo da capo tornando nella fase di attesa giocatori, mantenendo il round già preparato.

Se invece ci sono sempre giocatori connessi inizia il gioco.
Come successo per la pausa, viene settato l'alarm con la durata della partita (di default è settata a 180sec, ma può essere modificato dai parametri opzionali) e memorizzato il tempo di inizio.
Setta il bool della classifica a 0 così da impedire ai client di richiederla durante il gioco.
Viene poi inviata la matrice preparata durante la pausa e il tempo residuo di gioco a tutti i giocatori connessi e partecipanti al round. Come per la pausa sfrutto un while per simulare l'attesa della partita, che verrà interrotto dal cambio della variabile *pausa_gioco* da 0 in 1, effettuato dall'handler della *SIGALARM* allo scadere del tempo di gioco. Prima di rieseguire il ciclo da capo viene settato il bool del *round_pronto* a 0 così da notificare che la matrice preparata è stata utilizzata e ne va preparata una nuova.

Oltre ad aggiornare la variabile *pausa_gioco*, l'handler della *SIGALARM* si occupa anche di inviare il segnale *SIGUSR1* a tutti i giocatori che hanno partecipato al round notificandogli che devono caricare il risultato nella lista dei risultati. Viene infine creato lo *scorer* che si occuperà di generare la classifica. Se invece alla fine della partita non risultano giocatori connessi, è inutile chiamare lo *scorer* per creare la classifica, quindi si torna alla situazione di attesa giocatori come successo per la fase di pausa.

Thread scorer: all'avvio del thread viene innanzitutto svuotata la classifica vecchia e viene registrato il segnale *SIGUSR2* che ci servirà successivamente per notificare ai thread giocatori che possono prelevare la classifica e riprendere l'esecuzione.
Memorizzo il numero dei giocatori della partita precedente e creo un vettore di *risGiocatore* di dimensione *num_giocatori* che conterrà i risultati prelevati dalla lista *risList* tramite lo schema Produttore-Consumatore.

Infatti viene fatto un ciclo for che scorre il vettore di *risGiocatore* fino a quando non viene pienato. Durante il ciclo viene chiamata la mutex sulla *risList* e ci si mette in attesa se la lista è vuota, notificando ai thread tramite condition variables che stiamo aspettando che carichino i risultati. Una volta che la lista ha almeno un elemento, lo poppa e lo inserisce alla posizione *i* del vettore di *risGiocatore*. Viene poi rilasciata la mutex e ripetuto il ciclo.
Quando il vettore sarà pieno, si fa la *qsort* usando una funzione ausiliaria di comparazione che confronta il valore del campo punteggio di ogni risultato.
Ordinato il vettore si passa alla creazione della stringa in formato CSV seguendo il seguente standard: "*username1,punteggio1,username2,punteggio2,...*".
Una volta pronta la classifica, si aggiorna il suo bool e si invia il segnale *SIGUSR2* che invia la classifica a tutti i thread giocatori insieme al tempo rimanente all'inizio della prossima partita.
Si ritorna così alla fase di pausa del game e ricomincia il ciclo di gioco.
Notare come mentre il thread *scorer* stila la classifica, il thread *game* prepara il prossimo round.

Handler del SIGINT: quando ricevo un *SIGINT* da server, invece che scorrere la lista dei thread giocatori, scorro la lista *threadList* dei thread attivi, che comprende anche quelli non ancora registrati. Se la lista non è vuota, si procede a scorrere la lista e a mandare ad ogni thread un messaggio di chiusura facendogli presente che il server è stato chiuso. Viene poi chiuso il relativo thread.
Una volta chiusi tutti i thread vengono distrutte le liste, sia quella dei giocatori, che quella dei thread attivi e viene chiuso il socket.

STRUTTURAZIONE LOGICA DEL CLIENT

Il file *client.c* si occupa di comunicare con il giocatore attraverso lo *STDOUT* quando riceve qualcosa dal server o attraverso lo *STDIN* quando deve inviare qualcosa al server. A tal proposito anche il client è multithread.

Processo main: all'avvio tramite eseguibile, preleva i parametri dallo *STDIN*. Controlla i parametri *nome_server* e *porta* che serviranno per creare il socket di comunicazione con il server.
Vengono registrati i segnali *SIGINT* e *SIGUSR1* che ci serviranno per la corretta chiusura del client, e poi vengono creati i thread *sender* e *receiver* che si occuperanno rispettivamente di inviare comandi al server e di riceverli e stamparli a video correttamente.

Thread sender: inizia memorizzando il file descriptor passato come argomento della funzione. Successivamente inizia il loop condizionato dalla variabile volatile *run*. Ogni ripetizione del ciclo corrisponde ad una lettura da *STDIN*. La lettura viene memorizzata in un buffer e tramite la *strcmp* capiamo quale comando è stato inserito e cosa dobbiamo mandare al server. Per i comandi *registra_utente* e *p* abbiamo un secondo campo, quindi dovremmo andare a tokenizzare la prima volta fino allo spazio per eliminare il comando, la seconda volta fino al *'\n'* per trovare il campo data che andrà fornito al server per la completezza del comando. Se in questi due casi non viene inserito il secondo campo, allora si stampa in *STDOUT* un messaggio di errore.
Se il comando non è riconosciuto (non appartiene ai comandi disponibili) allora viene stampato in *STDOUT* un messaggio di errore e la stringa "[PROMPT PAROLIERE]-->".

Thread receiver: inizia memorizzando il file descriptor passato come argomento della funzione. Successivamente inizia il loop condizionato dalla variabile volatile *run*. Ogni ripetizione del ciclo corrisponde ad una ricezione di messaggio da server e la stampa in *STDOUT*. Infatti in base al tipo di *MSG* che viene ricevuto avviene una stampa particolare:

- nel caso di *MSG_MATRICE* si prende il campo data contenente la stringa della matrice e si stampa a video nella modalità grafica prescelta (in questo caso una vera e propria tabella)
- nel caso di *MSG_OK* e *MSG_ERR* si stampa direttamente il campo data poiché conterrà un avviso personalizzato notificato dal server
- nel caso di *MSG_PUNTI_PAROLA*, *MSG_TEMPO_ATTESA* e *MSG_TEMPO PARTITA* riceveremo nel capo data un valore che andremmo ad usare nella stringa di stampa da noi scelta.
- nel caso di *MSG_PUNTI_FINALI* riceveremo in formato CSV la classifica che andremo a tokenizzare e a stampare a video nella modalità grafica da noi prescelta.
- nel caso di *MSG_CHIUSURA_CLIENT* riceveremo il messaggio di chiusura da parte del server con annessa stringa contenuta in data che andremo a stampare a video, dopo verrà chiuso il socket.

Dopo ogni stampa a video verrà stampata la stringa “[PROMPT PAROLIERE]-->”.

Handler del SIGINT: quando si riceve un *SIGINT* da *STDIN*, si manda il segnale *SIGUSR1* al thread *sender* impedendogli così di inviare altri tipi di messaggio diversi da quelli inviati nell’handler del segnale. Si chiude il thread *receiver* che non dovrà più ricevere alcun dato dal server e si aspetta la terminazione del thread *sender*. Una volta fatto ciò si chiude il socket.

Handler del SIGUSR1: quando il thread *sender* riceve il segnale *SIGUSR1* stampa a video il messaggio di chiusura del gioco. Poi invia al server il messaggio di chiusura *MSG_CHIUSURA_CLIENT* notificandogli che deve cancellare tutto quello che ha su questo client. Infine termina il thread *sender* permettendo all’handler del *SIGINT* di continuare con la chiusura.

MAKEFILE

Il Makefile è usato per la compilazione separata dei file. Sono stati definiti i seguenti comandi per la compilazione e l’esecuzione:

- *make clean*, per pulire la directory dai file oggetto precedentemente creati
- *make* (o make all), per compilare il progetto
- *make server1025*, per eseguire il server passando solo i parametri “127.0.0.1” e “1025”
- *make server1025M,* per eseguire il server sfruttando i parametri dell’esempio precedente ma fornendo il file delle matrici “matrix.txt”
- *make server1026*, per eseguire il server passando i parametri obbligatori “127.0.0.1”, “1026” e i parametri opzionali “—matrici ./matrix.txt”, “—durata 4” e “—seed 20”
- *make client1025*, per eseguire il client su “127.0.0.1” e “1025”
- *make client1026*, per eseguire il client su “127.0.0.1” e “1026”