RELAZIONE PAROLIERE

Martini Matteo 636694

Il gioco del paroliere consiste nel trovare le parole contenute nella matrice che ci viene mostrata ad esempio nella matrice in figura contiene le parole Quasi, Equo, Equa, Bevo Vota ecc..

A	T	L	C
I	O	Qu	A
D	V	E	S
I	S	B	I

Pertanto il progetto consiste nel riprodurre il gioco con la specifica che in questa versione si possono indovinare solo parole lunghe almeno 4 e ci si può spostare sulla matrice solo in orizzontale e verticale.

Struttura del Progetto

Il progetto è composto da varie Cartelle:
-**Eseguibili** che contiene il Makefile per generare gi eseguibili di client e server
-**Sorgente** che contiene tutti i file sorgente, ossia i .c
-**Header** che contiene tutti i file header, ossia i .h
-**Text**, che contiene il dizionario da utilizzare in caso non ci venga dato dall'utente

Per ogni Sorgente è presente il corrispettivo Header eccezion fatta per Client e Server perchè essendo il corpo del progetto, si è preferito avere un file unico. Andremo di seguito ad analizzare i vari file di "libreria" creati per essere utilizzati nei file Client.c e Server.c

Stack

Le stutture dati create in questo file, come suggerito dal nome, sono delle Pile LIFO(Last-In First-Out) che verranno poi largamente utilizate all'interno del programma. Vengono definiti 3 tipi di Pile una per le stringhe che servirà a memorizzare le parole indovinate dall'utente, una per gli interi che andrà a memorizzare il file descriptor dei client collegati e l'ultima serve per memorizzare le informazioni legate al giocatore.
Oltre alle classiche funzioni dispongono anche di una Splice per andare a rimuovere uno specifico elemento, che si rivela indispensabile nel caso di Player_List e List come vedremo parlando del Server. Inoltre Player_List presenta delle funzioni che consentono di recuperare dalla lista: punteggio,username e file_descriptor di un determinato giocatore, che sfruttano un principio simile alla Splice che abbiamo nominato prima.
Ossia scandiscoono la pila finchè non trovano un elemento con il thread id desiderato e restituiscono quello che stavamo cercando, ad esempio la Player_Retrieve_Score restituisce il punteggio. Non vi è necessità di trovare il thread handler perchè è sufficiente chiamare pthread_self() per ottenere il thread_id da utilizzare per la ricerca delle informazioni relative all'utente. Questa ricerca è utilizzata anche nella struttura List per cercare il file descriptor sul quale sta avvenendo la connessione tra Server e Client.

```c
/*WORD_NODE*/
typedef struct w{
    char* word;
    struct w* next;
}Word_Node;

/*WORD_LIST*/
typedef Word_Node * Word_List;
```

```c
/*PLAYER_NODE*/
typedef struct pl{
    char* word;
    pthread_t handler;
    int points;
    int client_fd;
    struct pl* next;
}Player_Node;

/*PLAYER_LIST*/
typedef Player_Node* Player_List;

/*NODE*/
typedef struct n{
    int val;
    pthread_t thread;
    struct n* next;
}Node;

/*FD_LIST*/
typedef Node* List;
```

Matrix

Nel file Matrix.h viene definita una struttra dati per rappresenare un grafo, che verrà usata per la ricerca delle parole sulla matrice. Inoltre vengono definiti per semplicità i tipi booleani True e False.

```
typedef struct{
    // Numero di nodi nel grafo
    int V;
    //Array di caratteri per le etichette dei nodi
    char *nodes;
    //Lista di adiacenza
    int **adjList;
}Graph;
```

Il grafo viene costruito con nei nodi i singoli caratteri della matrice e gli archi sono le possibili direzioni per andare da un carattere all'altro. La ricerca delle parole avviene in DFS considerando la parola da cercare come un cammino tra le lettere che la compongono, ed in caso appartenga al grafo la parola sarà considerata valida.
Il file Matrix.c contiene tutte le funzioni necessarie a generare matrici leggendo da un file con la funzione Load_Matrix. Questa funzione legge una riga del file che ci viene indicato partendo dall'offset, anch 'esso indicato. Nel generare la matrice è stata posta particolare attenzione al caso della lettera Q che come da specifica è sempre seguita da una u minuscola, per questo è stato deciso di sostituirla con un carattere speciale per non complicare la rappresentazione della matrice che avviene mediante una stringa. Questa scelta rende anche riconoscibile la sequenza Qu consentendo quindi una corretta stampa all'utente.

Trie

Nel file Trie.h viene definita la struttura dati Trie che altro non è che un albero k-ario con 26 figli. Il campo is_word serve per indicare se siamo arrivati in un nodo che corrisponde ad una parola, questo perchè la struttura dati Trie è stata inventata per semplificare la ricerca di parole all'interno di un dizionario perchè ha costo pari alla lunghezza della parola.

```
typedef struct t{
    struct t* figli[NUM_CHAR];
    int is_word;
}Trie;
```

Questa struttura porta con sè un grande compromesso ossia, riduciamo il tempo richiesto per la ricerca delle parole aumentando così l'efficienza ma occupando una grande quantità di spazio perchè le parole di un dizionario sono numerose. Visto che viene già allocata una grande quantità di spazio poteva essere ragionevole ridurre il numero di step per convalidare le parole all'interno del programma creando 1 unico Trie con dentro tutte le parole componibili sulla matrice che poi verranno controllate sul file dizionario, però questo vorrebbe dire che per ogni matrice che viene proposta per il round è necessario creare un Trie con dentro approssimativamente 4720 parole che potrebbero avere o meno senso compiuto e quindi andrebbero controllate tutte sul dizionario ed al caso pessimo questa operazione costa circa O(n*4720) o qualcosa di simile dove n è il numero di parole del dizionario che come abbiamo detto prima è molto elevato, rendendo quindi difficile creare questo Trie in breve tempo.

Communications

La libreria Communications fornisce 3 funzioni di cui 2 sono speculari, ossia Receive_Message e Send_Message. La prima riceve il messaggio che è stato mandato dalla seconda, quindi le operazioni di read/write sono svolte nella medesima sequenza. Il primo passo consiste nell'invio della lunghezza del messaggio, per consentire al ricevente di predisporre lo spazio necessario, in seguito si manda il tipo del messaggio seguito dal contenuto effettivo.
L'ultima funzione definita in questo file consente di portare una stringa in UpperCase perchè è stato scelto di standardizzare le stringhe che vengono inviate tra Server e Client eccezion fatta per la registrazione utente per consentire di avere nomi anche in lowercase; quindi questa funzione molto semplicemente sottrae al carattere ASCII il valore 32 che rappresenta la differenza tra la prima lettera minuscola e la prima lettera maiuscola. Mentre nel File Communications.h sono definiti i tipi che i messaggi possono avere e tra questi ne sono stati aggiunti 3: MSG_PUNTEGGIO, MSG_CHIUSURA_CONNESSIONE ed MSG_SIGINT che vengono utilizzati rispettivamente per rispondere al comando score che può essere digitato dall'utente per ricevere il punteggio attuale durante la partita, comunicare al server che l'utente ha digitato il comando fine ed intende quindi chiudere la connessione, e per gestire la terminazione ccidentale come vdremo in seguito. Infine vengono definiti 3 messaggi che verranno inviati al client dal server per dargli il benvenuto, comunicargli il regolamento oppure elencare i comandi a lui disponibili.

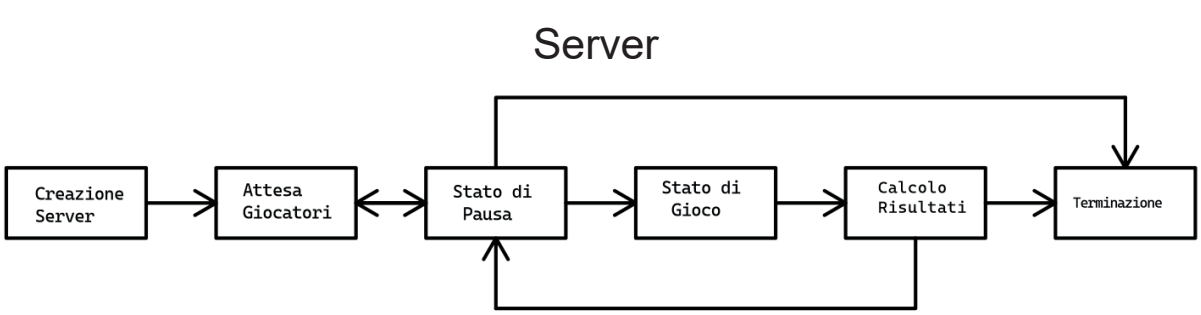
Client

Il file Client.c è la parte del gioco che si interfaccia con l'utente permettendogli così di giocare.
Questo file non utilizza delle particolari strutture dati perchè è stato scelto di rendere il Client il più lineare e semplice possibile, infatti il main definisce la connessione mediante il socket utilizzando i parametri presi da riga di comando, una sigaction per gestire i segnali e crea 2 thread che gestiranno le comunicazioni col Server il Merchant ed il Bouncer. Il primo si occuperà di riconoscere i comandi digitati dall'utente e successivamente di inviarli al Server se necessario, mentre il Bouncer dovrà ricevere tutti i messaggi che vengono inviati dal Server e questo generalmente avviene in risposta ad un messaggio mandato dal Merchant. Ci sono però alcuno casi in cui è il server ad inviarci autonomamente un messaggio, uno fra tutti quando il Server sta chiudendo la connessione mediante un SIGINT e bisogna gestire la chiusura; proprio per casi come questo il Client è stato pensato multi-thread perchè altrimenti si sarebbero dovute operare altre scelte per "sovvertire" il normale ordine delle cose che vede il Client come entità che invia messaggi ed il Server come entità che risponde soltanto impossibilitandolo quindi a mandare messaggi in autonomia.

Come accennato prima quando il Server decide di chiudere verrà inviato un messaggio al bouncer di tipo MSG_CHIUSURA_CONNESSIONE che farà comunicare al thread Merchant mediante un segnale SIGUSR1 di terminare immediatamente la comunicazione col server, andando poi a terminare. Questo restituirà il controllo al thread principale che dopo aver creato i thread era rimasto ad aspettare la loro terminazione che una volta avvenuta fara terminare il programma. Ci sono altri 2 modi per terminare il Client, ossia per scelta dell'utente digitando il comando fine che farà mandare al Merchant un messaggio MSG_CHIUSURA_CONNESSIONE verso il Server per comunicare la chiusura e prima di chiudere la connessione andrà ad avvisare mediante il segnale SIGUSR2 il Bouncer per consentirgli di terminare. Alla fine di questo procedimento il Merchant terminerà ridando quindi il controllo al main che come visto prima terminerà. Infine l'ultimo modo per terminare è mediante una disconnessione accidentale, che viene intesa come segnale SIGINT/SIGQUIT e questa gestione viene lasciata al thread principale, che dovrà innanzitutto terminare il Bouncer , in seguito invierà un segnale SIGUSR2 al merchant per comunicare la chiusura al Server ed andare quindi a chiudere la connessione dopo che il Merchant è terminato potendo così chiudere il programma.

```c
void gestione_terminazione_errata(int signum) {
    int retvalue;
    switch (signum){
        case SIGINT:
            if(pthread_self()== main_tid){
                printf("sigint\n");
                pthread_kill(merchant,SIGUSR2);
                pthread_cancel(bouncer);
                pthread_join(merchant,NULL);
                SYSC(retvalue,close(client_fd),"chiusura del client");
                /*chiudo il socket*/
                exit(EXIT_SUCCESS);
                return;
            }
            else return;
        case SIGQUIT:
            if(pthread_self()== main_tid){
                printf("sigint\n");
                pthread_kill(merchant,SIGUSR2);
                pthread_cancel(bouncer);
                pthread_join(merchant,NULL);
                SYSC(retvalue,close(client_fd),"chiusura del client");
                /*chiudo il socket*/
                exit(EXIT_SUCCESS);
                return;
            }
            else return;
```

```c
case SIGUSR1:
    pthread_exit(NULL);
    return;
case SIGUSR2:
    writef(retvalue,"Grazie per aver giocato\n");
    if (pthread_self()==merchant){
        Send_Message(client_fd,"chiudo",MSG_CHIUSURA_CONNESSIONE);
    }
    pthread_exit(NULL);
    return;
```



Una piccola premessa prima di addentrarci nell'analisi del comportamento del Server verrà fatta distinzione tra giocatori e client, dove i primi si intendono client che hanno effettuato la registrazione mentre i secondi devono ancora registrarsi.
Come descritto dal diagramma degli stati nella figura soprastante il server ha 6 stati di cui 3 vengono ripetuti finchè non viene scelto di terminare il programma. È importante notare come ci siano 2 modi per entrare nello stato di terminazione, dallo stato di pausa e dallo stato di Calcolo Risultati. Questa scelta è dovuta alla volontà di comunicare la classifica ai giocatori anche quando la partita viene interrotta da un SIGINT.
Passiamo ora ad una descrizione degli stati.

Creazione del Server
In questo stato avvengono 2 cose principali, vengono inizializzati i parametri passati da riga di comando, compresi quelli opzionali mediante la funzione getopt, dal main_thread il quale successivamente si occuperà di inizializzare tutte le strutture dati globali che verranno usate, come per esempio il dizionario che viene caricato in memoria mediante un Trie. Il main_thread verrà inoltre utilizzato come "Dealer" del gioco, ossia sarà lui a creare i round di gioco durante lo stato di pausa, per consentire questo è stato scelto di creare un thread che dovrà gestire le connessioni dei client e visto che si vuole ridurre il lavoro del main_thread sarà proprio il "Jester" a creare il socket. Il Jester ha quindi il compito di creare il socket lato server e mettersi in attesa delle connessioni alle cui risponderà creando un thread dedicato allo scambio di informazioni tra client e server. È stato deciso di predisporre un massimo di 32 client connessi contemporaneamente per giocare ed in caso si connettesse il 33esimo utente viene accettato ma visto che è stato raggiunto il limite verrà comunicato all'utente di dover aspettare a connettersi e successivamente la connessione sarà chiusa. È stato preferito chiudere la connessione immediatamente invece di lasciare l'utente in attesa perchè non si può sapere con esattezza quando si libererà un posto per poter giocare e quindi potrebbe avere un tempo di attesa molto lungo; questa convenzione inoltre ci dà un numero massimo di thread che possono essere attivi contemporaneamente, mentre in caso non si chiudesse la connessione potremmo potenzialmente averne infiniti.
Appena il Jester si mette in attesa delle connessioni si passa allo stato di Attesa Giocatori.

Attesa Giocatori
In questo stato il programma aspetta che un client connesso si registri mediante il comando registra_utente e la prima registrazione farà cambiare di stato verso lo stato di Pausa. La scelta di far inizare la pausa dopo la prima registrazione è stata pensata per dare la possibilità a più giocatori di connettersi prima che inizi il gioco in modo da garantire a tutti gli utenti lo stesso tempo di gioco, visto che sebbene sia possibile connettersi a partita in corso, il tempo rimanente non viene cambiato. Sebbene non mostrato esplicitamente dal diagramma è possibile ritornare in questo stato dallo stato di pausa in caso non ci siano più giocatori connessi, questo è stato scelto perchè si voleva evitare di "avviare" partite senza partecipanti.

Stato di Pausa
Questo è uno dei 3 stati che compongono il ciclo di gioco del programma. Nel Ciclo di gioco il passaggio di stato avviene mediante i segnali SIGALRM gestiti dal gestore dei segnali, apparte per lo stato di Calcolo Risultati ma lo vedremo meglio successivamente.
Qui vengono generate le matrici sui quali i client potranno giocare, ed il server in questo stato risponde solo ai messaggi di tipo MSG_PUNTI_FINALI, MSG_CHIUSURA_CONNESSIONE, MSG_PUNTEGGIO ed MSG_MATRICE, mentre tutti gli altri tipi vengono ricevuti ma ignorati. Lo stato di pausa ha una durata standard di 1 minuto che una volta trascorso fa cambiare di stato verso lo stato di gioco.

Stato di Gioco
Nello stato di gioco il Server risponde a tutti i tipi di messaggi, compiendo le dovute operazioni, per esempio se viene rilevato un messaggio di tipo MSG_PAROLA bisognerà prima di tutto verificare se la parola inserita dall'utente è presente nella matrice ed in caso affermativo cercarla nel dizionario. È stato scelto di preparare una copia dell'input ricevuto dall'utente per gestire il caso della lettera Qu visto che la parola dell'utente in caso non abbia la u dopo ogni Q non verrà convalidata per come sono state implementate le funzioni del file Matrix, mettendo quindi nella copia dell'input il carattere speciale presente nella matrice, ossia il "?".
Lo stato di gioco, qualora non venga specificato tramite parametro addizionale, ha una durata di 3 minuti nei quali vengono accettati tutti i messaggi dei vari client ed in caso abbiano tipo MSG_PUNTI_PAROLA si procederà a convalidare il payload che viene inviato. Il processo di validazione necessita di 3 controlli, ossia controllare la validità della parola nel dizionario, controllare che la parola suggerita dall'utente non sia già stata indovinata, ed infine controllare che sia componibile sulla matrice. Queste 3 operazioni avvengono in questa sequenza per provare a ridurre i costi computazionali, perchè cercare la parola nel dizionario ha un costo O(lunghezza parola), che al massimo può essere 16, grazie all'utilizzo della struttura dati Trie mentre la ricerca delle parole indovinate ha un costo variabile in base alla bravura dell'utente perchè al caso pessimo costa O(n) e con un utente mediamente bravo ha un costo simile alla ricerca sul trie quindi questi due step possono anche essere scambiati, e solo alla fine conviene verificare la presenza della parola sulla matrice perchè al caso pessimo ha costo O(n^2)visto che la ricerca della parola avviene in DFS.
Una volta passato il tempo di gioco si entra nello stato di Calcolo Risultati.

Calcolo Risultati
Questo stato ha lo scopo di stilare la classifica della partita appena trascorsa e ciò avviene mediante il thread Scorer che riceve i punteggio dai thread Gestori dei client mediante uno schema Produttore-Consumatore utilizzando una Coda LIFO(Last-In First-Out) utilizzando la mutua esclusione e le condition_variables.
Una volta recuperati tutti i punteggio lo scorer provvede a ordinarli mediante un QuickSort e memorizza la classifica cosi calcolata in una omonima variabile globale. Successivamente invierà un segnale a tutti i thread gestori per avvisarli di comunicare la classifica ai client e terminerà. Il thread scorer viene creato ogni volta che la si entra nello stato di Calcolo Risultati. Questo stato non è esplicitato nel codice perchè è considerabile come un sottostato dello stato di pausa perchè avviene in simultanea alla creazione del prossimo round. Se non è arrivato un segnale SIGINT il programma continuerà il ciclo di gioco ed entrerà quindi nello stato di pausa, altrimenti procederà a comunicare a tutti i client la chiusura del server per poi successivamente far terminare i thread gestori, ed infine verrà chiuso il socket del server. È interessante notare che in caso l'interruzione avvenga durante lo stato di gioco si passerà comunque per lo stato di Calcolo Risultati per terminare e questa scelta è stata intenzionale in modo da non "buttare" il risultato parziale degli utenti che in questo caso verranno prima avvisati della chiusura inaspettata e successivamente riceveranno la classifica prima che venga chiusa definitivamente la connessione.
Dal diagramma di stato si vede che anche dallo stato di pausa si può andare nella terminazione ed è ancora più immediato di quando avviene durante il Calcolo Risultati, perchè essendo la partita in pausa non sono state sottomesse parole e quindi tutti i client hanno punteggio 0, e questo ci consente di terminare, avvisando ovviamente i client, senza dover stilare la classifica.

Avviare il Progetto

Per avviare il progetto basta aprire il terminale, posizionarsi nella cartella Eseguibili ed utilizzare il comando make che farà generare gli eseguibili in 2 step, nel primo andrà a creare tutti i file oggetto utilizzando i file sorgente ed header di cui abbiamo parlato in precedenza e successivamente procederà a Linkare i file oggetto con le librerie necessarie a generare gli eseguibili chiamati Client e Server tramite i quali potremo per l'appunto eseguire il programma, ed inoltre eliminerà i file oggetto che risultano superflui una volta creato l'eseguibile.
Per semplificare l'esecuzione sono stati predisposti dei comandi nel makefile che avviano il server con vari argomenti opzionali da riga di comando che verranno elencati di seguito, con affianco il comando che eseguono:
-run-server: ./Server 127.0.0.1 20000 --matrici ../Text/Matrici.txt --diz ../Text/Dizionario.txt
-test-client: ./Client 127.0.0.1 20000
-run-server2: ./Server 127.0.0.1 30000 --durata 2 --seed 500
-test-client2: ./Client 127.0.0.1 30000

altri comandi definiti nel Makefile sono:
obj-clear: rm -f Server.o Client.o Matrix.o Stack.o Communications.o Trie.o
che rimuove i file oggetto.
clear: rm -f Server.o Client.o Matrix.o Stack.o Communications.o Trie.o Server Client
che rimuove sia i file oggetto che gli eseguibili

Inoltre sono presenti dei programmi di test per testare le funzionalità del progetto, è sufficente posizionarsi nella cartella test_header e digitare make, di seguito verranno elencati i vari test:
-./test_trie avvia un test per verificare il corretto riconoscimento delle parole sul trie
-./test_matrix avvia un test per verificare il corretto caricamento delle matrici tramite file ed il riconoscimento delle parole
-./test_comms avvia un test per verificare il funzionamento di Send_Message e Receive_Message

È possibile inotlre testare rapidamente il funzionamento dello scorer, sebbene non sia predisposto un file apposito. Per fare questo è necessario aprire il file Server.c ,andare nella riga 34 ed impostrae DURATA_PAUSA a 2 o ad un qualsiasi valore piccolo che saranno i secondi della pausa, successivamente basta compilare con make, dopo essersi posizionati nella cartella Eseguibili e per avviare il server è sufficiente utilizzare il comando:
-run-scorer: ./Server 127.0.0.1 40000 --durata 0.00278
-test-scorer: ./Client 127.0.0.1 40000

questo test fa eseguire il programma con una durata limitata della partita per evidenziare il lavoro dello scorer e ridurre il tempo pausa aiuta a visualizzarlo più in fretta, è anche possibile fare un CTRL-C durante il gioco per vedere la tereminazione in caso di Stato di Gioco. La volontà di mostrare la terminazione è il motivo principale dietro alla scelta di non predisporre un file di test dedicato allo scorer.

Scelte Aggiuntive

Questa sezione serve per spiegare meglio delle scelte di Design la cui spiegazione non ha trovato spazio in precedenza, ad esempio come avvengono le transizioni di stato quando ci troviamo nel ciclo di gioco. Queste transizioni sono gestite tutte dal gestore di segnali, perchè avvengono mediante un SIGALRM che una volta rilevato in base alla variabile game_on che rappresenta lo stato del server capisce cosa deve fare:
-Se siamo in stato di gioco, quindi game_on = 1, bisogna passare allo stato di Calcolo Risultati ed in seguito a quello di pausa
-Se siamo in stato di pausa, quindi game_on = 0, bisogna passare allo stato di Gioco
Lo stato di calcolo risultati non è esplicitato in questa gestione poichè ha una durata davvero ridotta rispetto al minuto standard della pausa, anche con il massimo numero di client connessi.
Un'altra precisazione riguarda la terminazione in caso la partita sia in corso, in quel caso non si accettano più connessioni e soprattutto non si permette ai client di chiudere la connessione finchè non è il server a deciderlo. Questa scelta è abbastanza critica perchè potrebbe portare a delle incongruenze con lo scorer, in quanto se un utente decidesse di disconnettersi il messaggio MSG_CHIUSURA_CONNESSIONE non arriverebbe mai al server dato che non si accettano più messaggi ma si spediscono soltanto, e soprattutto non c'è bisogno che l'utente si disconnetta prima del tempo perchè se il server deve terminare l'esecuzione da li a poco l'utente si disconnetterebbe in automatico. La lista dello scorer è stata implementata come Word_List perchè deve contenere soltanto una stringa in formato CSV con nome utente e punteggio cosicchè una volta recuperata dalla coda sia sufficiente tokenizzare sulle virgole per ottenere i campi del messaggio.
Infine nel progetto viene usata anche una libreria macro.h per controllare i valori di ritorno di tutte le SYSTEM CALL utilizzate all'interno del progetto.