

Il Paroliere

Matteo Cherubini 658274

Contents

1	Strutture dati	1
1.1	Trie	1
1.2	Array dinamico	2
1.3	Implementazione	2
1.4	Dati di gioco	2
1.5	Messaggi	2
2	Struttura del codice	3
3	Struttura e funzionamento dei programmi	4
3.1	Server	4
3.2	Client	5
3.3	Protocollo di comunicazione	6
4	Testing del programma	7
4.1	Debugging con GDB	7
4.2	Controllo dei parametri	7
4.3	Verifica del funzionamento del gioco	8
4.4	Test sulla macchina Laboratorio II	8
4.5	Test su indirizzo pubblico	8
4.6	Memory Leak	8
5	Compilazione e utilizzo	10

1 Strutture dati

La prima scelta durante la fase di progettazione è stata quella di capire quali strutture dati fossero più adeguate per gestire le informazioni di gioco.

1.1 Trie

Il Trie è un tipo di struttura dati ad albero ottimale per salvare un grande insieme di stringhe. Ho deciso di utilizzare questa struttura dati per caricare in memoria il contenuto del file dizionario. Il file del dizionario contiene migliaia di parole, e in questo caso la struttura del **Trie** garantisce un'ottima velocità di ricerca all'interno del set (al peggio logaritmica) e non si comporta male nella

gestione dell'allocazione di memoria (anche se, pur essendo presenti migliaia parole, molto spazio rimane lo stesso occupato da puntatori **NULL**).

1.2 Array dinamico

Avevo bisogno di una struttura dati per salvare l'insieme delle parole trovate dai giocatori. Tuttavia il **Trie** risulta poco efficiente nell'allocazione di memoria per un piccolo set di stringhe, quindi la scelta è ricaduta su un **Array dinamico**. Il tempo di ricerca è lineare e non logaritmico, ma le parole trovate all'interno di una partita sono poche in numero, quindi risulta una scelta implementativa semplice ed adeguata. Inoltre, ho potuto riutilizzare questa struttura dati anche per caricare la lista delle matrici da file e per gestire la lista dei punteggi con qualche piccola aggiunta alla *struct*.

1.3 Implementazione

Entrambe queste strutture dati sono state implementate nel codice con delle *struct*, in dei file appositi che contengono le relative funzioni per l'allocazione della memoria, creazione e popolamento dei dati, gestione e ricerca e liberazione della memoria. Nel caso del **Trie**, le funzioni utilizzano spesso la ricorsione. Inoltre, per stilare la classifica, ho implementato un semplice algoritmo di **Bubble Sorting** (sufficiente dato il numero irrisorio di giocatori) per riordinare l'**Array dinamico** dei punteggi.

1.4 Dati di gioco

Per salvare i dati di gioco ho scelto di allocare dinamicamente spazio in memoria ordinandolo con delle *struct*. In particolare, lato server, è presente una *struct* principale **GameInfo** che contiene tutte le informazioni di connessione, preferenze dell'utente e puntatori in memoria ai dati letti dai file all'avvio del programma (dizionario ed eventuali matrici). Inoltre è presente un puntatore ad una *struct* **GameSession** che contiene le informazioni sulla sessione di gioco corrente, come la lista dei **Player** (definiti da un'ulteriore *struct*), la fase di gioco e la matrice di gioco attuale.

1.5 Messaggi

L'ultima struttura dati fondamentale è quella utilizzata per inviare e ricevere messaggi tra *Client* e *Server*. La scelta è stata influenzata dalla necessità di allocare solamente lo spazio necessario per ogni messaggio, e non una dimensione fissa. Ho utilizzato una *struct*, il cui primo elemento è la **lunghezza dei dati** del messaggio, il secondo è il **tipo del messaggio** e il restante spazio di allocazione è riservato ai **dati**, la cui dimensione è appunto descritta dal primo elemento della *struct*. Così facendo, risulta immediato capire quanti *byte* bisogna scrivere o leggere durante l'invio o la lettura dei messaggi. Ho scritto due funzioni apposite per wrappare queste istruzioni ed utilizzarle sia nel *Server*

che nel *Client*, oltre a funzioni apposite per estrapolare i dati per inviarli.

Il dettagli sul protocollo messaggi hanno una sezione dedicata più avanti

2 Struttura del codice

- **Folder** [Legenda]
 - Project files
 - * Compilation dependencies
- **Main:** [Main files per gestione del server e del client e dei relativi threads]
 - Server.c
 - * GameState
 - * CommsHandler
 - * DynamicArray
 - * Trie
 - * Utility
 - Client.c
 - * CommsHandler
 - * Utility
- **GameState:** [Gestione del gioco e delle relative strutture dati]
 - GameState.c
 - GameState.h
- **DataStructures:** [Gestione delle strutture dati Trie e Dynamic Array]
 - Trie.c
 - Trie.h
 - DynamicArray.c
 - DynamicArray.h
- **CommsHandler:** [Gestione protocollo di comunicazione Client/Server]
 - CommsHandler.c
 - CommsHandler.h
- **Utility:** [Funzioni di Utility generiche]
 - Utility.c
 - Utility.h

- **Data:** [Database dizionario e matrici]
 - dizionario.txt + dizionario_small.txt
 - matrici.txt

Tutte le librerie utilizzate appartengono allo **standard C POSIX**.

Oltre alle classiche librerie per la gestione della memoria e delle chiamate di sistema, ho utilizzato *string.h* e *ctype.h* per la gestione delle stringhe e dei caratteri, *getopt.h* per leggere i parametri dell'eseguibile, *signal.h* per la gestione dei segnali, *stdbool.h* per utilizzare i *booleani* invece di 0 e 1 e *netdb.h* e *arpa/inet.h* per la gestione e il controllo degli indirizzi IP.

3 Struttura e funzionamento dei programmi

3.1 Server

1. **main**

Una volta verificati e decodificati i parametri in ingresso obbligatori e opzionali, inizializza le strutture dati, carica in memoria il contenuto dei file e crea il thread **startServer**. Infine, rimane in attesa del segnale *SIGINT*, con conseguente attesa della terminazione del thread **startServer** e liberazione della memoria occupata dalle strutture di gioco tramite la funzione **freeGameMem()**.

2. **startServer**

Si occupa di creare il canale di comunicazione con i *Client* e di accettare nuove connessioni. Ad ogni connessione alloca in memoria un nuovo *Player* e lo passa come parametro al thread **handlePlayer** appena creato. Inoltre, appena creato il socket di comunicazione, avvia il timer delle fasi di gioco con *alert()*. Il thread rimane in ascolto per nuove connessioni finché non viene terminato dal **main** (*terminazione indotta dalla chiusura del socket*).

3. **handlePlayer**

Crea due nuovi thread, **playerSync** e **playerAsync** che si occupano di gestire le comunicazioni con il *Client*. Rimane poi in attesa della terminazione di **playerSync** (data dalla disconnessione del *Client* stesso, sia che avvenga in modo accidentale, sia che voluto dal *Client* o indotto dal **main** alla terminazione del programma). Una volta terminato **playerSync**, il giocatore viene de-registrato e viene indotta la terminazione di **playerAsync** con un segnale ed infine, tramite **deletePlayer**, il thread si occupa di liberare la memoria occupata dal giocatore e di liberare lo slot nella partita e nella lobby se necessario.

4. **playerSync**

Si occupa di leggere i messaggi provenienti dal *Client* e rispondere di conseguenza. Lo scambio di messaggi avviene tramite il protocollo di comunicazione definito in **CommsHandler** e nel testo del progetto. In breve, le principali richieste sono quella di registrazione che, una volta verificata, aggiunge il giocatore alla partita, sincronizzandosi con gli altri thread grazie al lock *player_mutex*, e quella di ricezione parola, che porta alla verifica della sua presenza prima nella matrice di gioco e poi nel dizionario. La terminazione di questo thread è indotta dalla chiusura del canale di comunicazione con il *Client* che interrompe la lettura dei messaggi, oppure da una richiesta esplicita dal *Client* tramite il protocollo di comunicazione.

5. **playerAsync**

Rimane ciclicamente in attesa del segnale *SIGUSR1*, inviato da **handlePlayer** o da **scorer**, dopo il quale, se la partita sta terminando entra in una sezione critica gestita dal lock *score_mutex*. In questa sezione, aggiunge lo score del giocatore alla lista condivisa con i thread degli altri giocatori e rimane in attesa sulla variabile di condizione *score_ready_cond*. Una volta svegliato dallo **scorer**, invia al *Client* la classifica. Se la partita sta ricominciando, invece, si occupa solo di inviare le informazioni al giocatore. Ad ogni segnale *SIGUSR1* controlla se il giocatore è ancora registrato, e se non lo è termina l'esecuzione.

6. **timeTick**

Handler per il segnale *SIGALRM* inviato per la prima volta dal thread **serverThread**. Decrementa il tempo rimanente alla prossima fase di gioco e, se il tempo è terminato, si occupa di far procedere il gioco alla fase successiva, generando la nuova matrice e resettando gli score all'inizio di ogni partita. Infine avvia il thread **scorer**, attendendone la terminazione prima di inviare un nuovo segnale *SIGALRM* con la durata di 1 secondo.

7. **scorer**

Thread caratterizzato da un'unica sezione critica che racchiude tutte le sue funzionalità. Si occupa di resettare gli score dei singoli giocatori e di inviare un segnale *SIGUSR1* ai **playerAsync**, per poi attendere che abbiano tutti finito di aggiungere il proprio score alla lista condivisa. Una volta arrivata l'ultima *signal()*, ordina la lista per creare una classifica, estrapola il vincitore e segnala ai **playerAsync** in attesa che possono inviare le statistiche ai rispettivi giocatori.

3.2 Client

1. **main**

Una volta verificati e decodificati i parametri in ingresso, crea il socket di comunicazione con il *Server*, crea il thread **readBuffer** ed entra nel **Game Loop**, una sezione critica che garantisce la mutua esclusione dell'I/O del

terminale. Una volta preso il lock *output_mutex*, crea il thread **readInput** e rimane in attesa di un risultato. Se il risultato è un comando valido, invia al *Server* il corrispettivo comando e rimane in attesa sulla variabile di condizione *output_available* finché il *Server* non risponde, per poi riproporre l'input all'utente. Se invece la lettura è stata interrotta dal **readBuffer**, significa che è arrivato un messaggio asincrono dal *Server* e quindi disabilita l'input all'utente attendendo che il messaggio venga gestito. In entrambi i casi la *signal()* viene inviata da **readBuffer**. La terminazione avviene quando la flag **bGameLoop** viene settata a *false* dal **readBuffer** o da una richiesta esplicita dall'utente.

2. **readBuffer**

Attende ciclicamente la ricezione di un messaggio da parte del *Server*. Ad ogni messaggio ricevuto, invia un segnale *SIGUSR1* al thread **readInput** per interrompere l'input dell'utente, acquisisce il lock *output_mutex* (per assicurarsi di essere l'unico ad utilizzare l'I/O del terminale) e gestisce il messaggio ricevuto dal *Server* (secondo il protocollo di comunicazione definito in **CommsHandler**), segnalando poi al **main** la terminazione con una *signal()*. Se la lettura del messaggio fallisce, il socket di comunicazione con il *Server* è stato chiuso. Quindi setta la flag **bGameGoing** a *false* e termina, causando la terminazione del **main**.

3. **readInput**

Presenta il prompt all'utente e attende l'input per poi comunicarlo al **main** e terminare. All'arrivo di un messaggio da parte del *Server*, il thread riceve un segnale *SIGUSR1* che interrompe l'input dell'utente e termina. Questo per evitare che l'utente possa inviare altri comandi mentre il **readBuffer** gestisce il messaggio ricevuto dal *Server*. Il thread verrà poi riavviato dal **main** se il programma non sta terminando.

4. **signalHandler**

Handler per il segnale *SIGINT*, che porta alla chiusura del socket di comunicazione, quindi alla terminazione del **readBuffer** e di conseguenza del programma (in modo corretto), e per il segnale *SIGUSR1*, utilizzato nel caso di ricezione di un messaggio dal *Server* come descritto sopra.

3.3 Protocollo di comunicazione

Il protocollo di comunicazione è definito da un insieme di caratteri che indicano la tipologia del messaggio (**type**). Ho suddiviso i messaggi in *messaggi di testo* e in *messaggi numerici*. In base al **type**, i *Client* e il *Server* sapranno se dovranno interpretare i dati del messaggio come valore testuale o numerico. Oltre ai **type** descritti nel testo del progetto, ne sono stati aggiunti 3 ulteriori:

- **MSG_CLOSE_CLIENT:**

Usato per inviare al *Server* un'esplicita richiesta di terminazione del *Client*

- **MSG_VINCITORE:**
Usato per inviare ai *Client* il nome del vincitore della partita oltre alla *Scoreboard*
- **MSG_PUNTI_PERSONALI:**
Usato per inviare il punteggio personale del giocatore ai *Client* durante la partita, distinto da **MSG_PUNTI_FINALI** utilizzato esclusivamente per la *Scoreboard*

Ho inoltre deciso di ridurre al minimo indispensabile la quantità di logica nel *Client* per il controllo della validità dei messaggi durante le diverse fasi della partita. L'unico controllo che esegue è quello di validità del comando e della relativa sintassi (cioè il numero dei parametri che lo seguono) al fine di poter inviare un messaggio corretto al *Server*. Tutti i casi restanti, come la validità del nome inserito in fase di registrazione e dei comandi utilizzati prima o dopo la registrazione o nelle diverse fasi della partita vengono controllate dal *Server* che, in caso di errore, invierà al *Client* un **MSG_ERR** con la descrizione dell'errore nel campo **data**.

4 Testing del programma

- Debugging con il tool **GDB**
- Controllo dei parametri in input ai file eseguibili
- Verifica effettivo funzionamento del gioco *Il Paroliere*
- Test sulla macchina **Docker lab2**
- **Port Forwarding** e test con indirizzo pubblico
- Controllo dei **Memory Leak** con il tool **Valgrind**

4.1 Debugging con GDB

Durante la fase di debugging è stato utile il tool **GDB** per risolvere gli errori che si sono presentati e per verificare la correttezza dei dati scambiati tra *Server* e *Client* con l'utilizzo di *breakpoints*.

4.2 Controllo dei parametri

La prima parte dei test è stata dedicata alla corretta lettura dei parametri, in modo particolare quelli opzionali relativi al *Server*. Il controllo del corretto ordinamento di quelli obbligatori, la corretta lettura e conversione di tipo dei parametri numerici e la verifica del corretto utilizzo della funzione **getopt()** per leggere i parametri *double-dash*.

4.3 Verifica del funzionamento del gioco

La fase di testing del corretto funzionamento del gioco è stata la più importante. In particolare, il testing ha riguardato la corretta verifica della presenza della parola nella matrice e nel dizionario, e il corretto funzionamento dei casi in cui la parola fosse formata dalla combinazione di caratteri *Qu*. Inoltre è stato importante verificare l'assenza di eventuali *Race Conditions* e il corretto funzionamento del parallelismo e della sincronizzazione tra thread (ad esempio quando più client si registrano al gioco contemporaneamente).

Un test di verifica del funzionamento del gioco è riportato nella sezione 5

4.4 Test sulla macchina Laboratorio II

Test del **Makefile** (quindi della corretta compilazione) e dei programmi sulla macchina di *Laboratorio II* avviata con **Docker**. Lo sviluppo è comunque avvenuto in ambiente **UNIX**, ma valeva la pena fare un'ulteriore verifica.

4.5 Test su indirizzo pubblico

L'ultimo test del corretto funzionamento della comunicazione su rete è stato eseguito facendo **Port Forwarding** sul modem e avviando il server sulla porta inoltrata, facendo connettere il *Client* all'indirizzo pubblico corrente.

4.6 Memory Leak

In C è fondamentale liberare la memoria precedentemente allocata dopo aver terminato di usarla. Il tool **Valgrind** consente di verificare che tutta la memoria allocata sia stata liberata alla terminazione del programma. Alla pagina successiva sono riportati gli output di Valgrind su tutti i metodi di terminazione possibili del *Server* e del *Client*

*NOTA: dai test risulta un unico caso in cui **valgrind** rileva una porzione di memoria non liberata. Dopo aver verificato attentamente che il codice scritto liberasse correttamente la memoria allocata, una ricerca sul web ha portato alla conclusione che la funzione di libreria apposita **freeaddrinfo()** di **netdb.h** non liberi tutta la memoria allocata dalla funzione **getaddrinfo()**, utilizzata in **Utility.c** per verificare la validità dell'hostname inserito come parametro, e che non ci siano altri metodi per farlo. Questo leak avviene solo quando il nome del server passando come parametro non sia un **IPv4**, dato che in quel caso la verifica dell'hostname non avviene.*


```

^C
Interruzione del gioco in corso...
==33168==
==33168== HEAP SUMMARY:
==33168==   in use at exit: 0 bytes in 0 blocks
==33168== total heap usage: 580,631 allocs, 580,631 frees, 125,382,924 bytes allocated
==33168==
==33168== All heap blocks were freed -- no leaks are possible
==33168==
==33168== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figure 1: Output del *Server* alla terminazione con *CTRL+C*

```

[PROMPT PAROLIERE]—>
==33987==
==33987== HEAP SUMMARY:
==33987==   in use at exit: 0 bytes in 0 blocks
==33987== total heap usage: 42 allocs, 42 frees, 7,955 bytes allocated
==33987==
==33987== All heap blocks were freed -- no leaks are possible
==33987==
==33987== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figure 2: Output del *Client* alla terminazione del *Server*

```

[PROMPT PAROLIERE]—>fine
==33645==
==33645== HEAP SUMMARY:
==33645==   in use at exit: 0 bytes in 0 blocks
==33645== total heap usage: 91 allocs, 91 frees, 9,386 bytes allocated
==33645==
==33645== All heap blocks were freed -- no leaks are possible
==33645==
==33645== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figure 3: Output del *Client* alla terminazione con il comando *fine*

```

[PROMPT PAROLIERE]—>^C
==34330==
==34330== HEAP SUMMARY:
==34330==   in use at exit: 0 bytes in 0 blocks
==34330== total heap usage: 81 allocs, 81 frees, 9,735 bytes allocated
==34330==
==34330== All heap blocks were freed -- no leaks are possible
==34330==
==34330== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figure 4: Output del *Client* alla terminazione con *CTRL+C*

5 Compilazione e utilizzo

Nei file del progetto è presente un **Makefile** che automatizza la procedura di compilazione dei file sorgente. Il target default del **Makefile** genera entrambi gli eseguibili **paroliere_srv** e **paroliere_cl** tramite l'apposito comando *make*. Il comando deve essere eseguito dalla cartella più esterna del progetto, dove è presente il **Makefile**. Gli eseguibili verranno generati nella stessa cartella.

Esempio di esecuzione del *Server*:

```
./paroliere_srv 127.0.0.1 6000 -matrici Data/matrici.txt -durata 5
```

Esempio di esecuzione del *Client*:

```
./paroliere_cl 127.0.0.1 6000
```

Test di verifica dal Client

Dopo essersi registrati con il comando registra_utente, è possibile verificare il funzionamento del gioco utilizzando la prima matrice presente nel file Data/matrici.txt e le parole CASI e QUASI inviate con il comando p. La prima parola è presente nel file dizionario Data/dizionario.txt, mentre la seconda è presente in entrambi e contiene la combinazione di lettere Qu. Altre parole presenti nella matrice sono SI e BEVO e sono validate solo dal dizionario Data/dizionario.txt

Per ulteriori informazioni sulle meccaniche di gioco, utilizzare il comando aiuto