The “Real” Paroliere online

Progetto Finale del corso di Laboratorio 2 A

versione 1.1 (con addendum di non superamento di prove in itinere)

Tae Hwan Kim

Matricola 614436

1. Architettura del Sistema

Il progetto è suddiviso in due macro-componenti:

- Server: Responsabile di
  - Gestione della logica del gioco
  - Orchestrazione delle partite (inizio e fine della partita/pausa tra partite),
  - Comunicazione con i client
  - Sincronizzazione di punteggi.
- Client:
  - fornisce un’interfaccia interattiva per l’utente
  - gestisce la comunicazione con il server
  - visualizza in tempo reale aggiornamenti e risultati di gioco

tale suddivisione permette isolamento di funzioni, rendendo il sistema piu’ modulare e mantenibile.

---

2. Principali scelte del progetto

2.1. Assenza di header file

Durante la fase di sviluppo, si è scelto di evitare la creazione di header file separati, considerando che il progetto ha dimensioni relativamente contenute. Questa scelta è stata motivata dal voler mantenere la semplicità e ridurre la complessità complessiva del codice. In particolare, l'assenza di header file ha permesso di semplificare i passaggi necessari per modificare e testare il codice, rendendo il flusso di lavoro più snodato ed efficiente.

2.2. Strutture dati e algoritmi

- **Dizionario:**

  il dizionario e’ implementato, come consigliato nelle specifiche del progetto, tramite struttura dati Trie che consente ricerche in tempo O(L), L sta per lunghezza di parola

  - Dettagli implementativi:
    - e’ implementato utilizzando lista concatenata, e ogni nodo contiene un flag `end_of_word` che indica se rappresenta la fine di una parola e un array di puntatori ai figli.

        - la gestione della sequenza “Qu” viene effettuato verificando che, in presenza della lettera “q”, l’elemento successivo sia “u”; in tal caso, come richiesto, token “Qu” viene trattato come unico elemento.

- **Matrice di gioco e DFS**

  la matrice 4×4 e’ gestita come un array lineare di 16 elementi

  - Dettagli implementativi:

    - l’esistenza della parola proposta nell matrice viene controllata dalla funzione `is_word_in_matrix` che all’interno di essa tokenizza la parola proposta (trattando “Qu” come un singolo token) e utilizza un algoritmo di Depth-First-Search (DFS) implementato all’interno della funzione `dfs_find` esamina la matrice per verificare che le lettere siano effettivamente collegate tramite celle contigue in tutte le direzioni (orizzontale, verticale e diagonale)
    - l’algoritmo implementato assicura che ogni cella venga utilizzata al massimo una volta per parola, marcandola `visited` , cosi’ da garantire la correttezza di validazione

- **Ordinamento dei punteggi:**

  Al termine di ogni partita, i punteggi raccolti vengono ordinati usando un algoritmo di ordinamento semplice ovvero Bubble sort, algoritmo scelto considerando numero massimo di client e’ piccolo (32) che ha minore overhead di memoria visto che non utilizza stack ausiliari come Quick sort o Merge sort

2.3. Logica di comunicazione

La comunicazione tra il server e i client avviene tramite socket TCP/IP utilizzando il protocollo `AF_INET` con socket di tipo `SOCK_STREAM` . Permettendo una connessione affidabile e bidirezionale tra server e client.

La gestione dei messaggi tra i due componenti è basata su un protocollo custom che si struttura in tre parti fondamentali:

1. **type (1 byte)**: Indica il tipo di messaggio, utile per determinare l'azione da intraprendere (ad esempio, registrazione, login, punteggio, ecc.).
2. **length (4 byte)**: Specifica la lunghezza del campo "data" in **network byte order**, utilizzando la funzione `htonl()` per garantire la corretta conversione tra il formato host e quello di rete.
3. **data (variabile)**: Contiene i dati effettivi del messaggio, come il nome dell'utente o la matrice di gioco, a seconda del tipo di messaggio.

Ogni messaggio viene inviato dal server al client (e viceversa) tramite funzioni `send_message()` che scrive i dati nel socket, e ricevuto dal server tramite `receive_message()` che gestisce la lettura dei messaggi e la loro decodifica.

2.2.1. Gestione del timeout e della disconnessione

Ogni client ha un **timeout di inattività** configurato tramite l'opzione `SO_RCVTIMEO` nel socket. Se il client non invia dati per un periodo predefinito, la connessione viene chiusa dal server. In caso di disconnessione per inattività o errore, il server invia un messaggio `MSG_SERVER_SHUTDOWN` per notificare la chiusura.

2.4. Gestione Concorrenza

2.4.1. Sincronizzazione: Mutex e Condition Variables

- **Fine-grained Mutex:**
  - `clients_mutex` : protegge l'array dei client e le operazioni che aggiornano lo stato dei client (registrazione, punteggi, disconnessione)
  - `registered_mutex` : isola l'accesso alla struttura degli utenti registrati, gestendo in modo sicuro le operazioni di registrazione e cancellazione
  - `log_mutex` : garantisce che le scritture sul fil di log siano serializzate
  - `score_queue_mutex` e `bacheca_mutex` : proteggono, rispettivamente, la coda dei punteggi e la struttura dati della bacheca
  - `ranking_mutex` : coordina il flusso tra thread Scorer e il thread Orchestrator insieme a `ranking_cond`assicurando che la classifica finale venga comunicata prima dell'avvio della partita successiva
  - `console_mutex` : protegge l'output sulla console di server (`stdout`), prevenendo interleaving di messaggi quando piu' thread stampano simultaneamente
  - `output_mutex` : protegge le operazioni di output sulla console di client, garantendo che i messaggi non vengano mescolati quando piu' thread stampano contemporaneamente
- **Condition variables:**
  - `score_queue_cond` : il thread Scorer attende che il numero di punteggi raccolti raggiunga il numero atteso, evitando busy-waiting e ottimizzando l'uso della CPU
  - `ranking_cond` : coordina il flusso tra thread Orchestrator e il thread scorer insieme a `ranking_mutex` assicurando che la classifica finale venga comunicata prima dell'avvio della partita successiva

2.4.2. Gestione dei thread

Thread sul lato client

- **client_receiver:**

  Un thread dedicato alla ricezione continua dei messaggi dal server, garantisce che l'interfaccia rimanga reattiva
  - Riceve i messaggi con la funzione `receive_message()` , li interpreta e gestisce eventuali casi di errore/timeout impostando un flag di terminazione (`shutdown_flag`)
  - L'output e' protetto dal mutex `output_mutex` per evitare conflitti con il thread principale (input utente)

Thread sul lato server

- **client_thread**
  - Uno per ogni client connesso, viene creato un thread e associato al client al momento in cui viene accettata una nuova connessione di un nuovo client. Gestisce la comunicazione bidirezionale (richieste/risposte) e aggiorna lo stato del client (login, punteggi, ecc.)
- **orchestrator_thread**
  - Gestisce il ciclo partita/pausa. All'avvio di ogni partita, rigenera la matrice (da file se specificato o causalmente chiamando funzione `generate_matrix` , resettando lo stato dei client

(punteggi, parole gia' inviate) e notificando l'inizio della partita.

- **scorer_thread**
  - Raccoglie i punteggi inviati dai thread client in una coda condivisa, li ordina e costruisce la classifica finale in formato CSV. La comunicazione di tale classifica avviene poi tramite l'invio di messaggi ai client

---

3. Strutturazione del codice

Il progetto e' strutturato in vari file sorgenti.

```
paroliere/
├── Makefile
├── src/
│   ├── server/
│   │   ├── server_main.c
│   │   ├── server_paroliere.c
│   │   ├── dictionary.c
│   │   ├── matrix.c
│   ├── client/
│   │   ├── client_main.c
│   │   ├── client_paroliere.c
├── resources/
│   ├── dictionary.txt
│   ├── matrix.txt
└── paroliere.log (file di log generato dal server)
```

3.1. Divisione in moduli

Server:

- `server_main.c`
  - responsabile dell'inizializzazione del server, come parsing dei parametri CLI, configurazione del socket, gestione dei segnali come SIGINT per lo shutdown
- `server_paroliere.c`
  - implementa logica centrale di gioco
  - gestisce i thread dedicati ai client, il ciclo partita/pausa, e la sincronizzazione tra thread (per esempio, l'invio della classifica finale)
  - include la logica per inviare messaggi, notificare i client in caso di timeout o errori e coordinare l'attivita' di gioco
- `dictionary.c`
  - implementa il trie, con funzioni per creare nodi, inserire parole, e verificare l'esistenza di parola e corretta liberazione di memoria
- `matrix.c`

- gestisce la generazione della matrice di gioco, sia in modo casuale (utilizzando `rand()` e un seed configurabile) sia tramite lettura da file
- implementa validazione delle parole proposte tramite DFS

Client:

- `client_main.c`
  - gestisce la connessione al server (risoluzione hostname, apertura del socket) e avvia il thread di ricezione
- `client_paroliere.c`
  - fornisce l'interfaccia utente interattiva per l'invio dei comandi
  - gestisce un thread separato per ricevere visualizzare i messaggi provenienti dal server, mantenendo aggiornato il prompt interattivo

3.2. Librerie e API utilizzate

ne cito alcuni di quelle che ho utilizzato all'interno del progetto.

- **Librerie standard C:**
  - `<stdio.h>` , `<stdlib.h>` , `<string.h>` per I/O gestione della memoria.
- **Librerie POSIX**
  - `<pthread.h>` per la gestione dei thread
  - `<sys/socket.h>` , `<arpa/inet.h>` , `<netinet/in.h>` per la comunicazione di rete
  - `<signal.h>` per la gestione dei segnali

---

4. Struttura dei programmi sviluppati

4.1. Server

- **Fasi operative:**
  1. **Avvio e inizializzazione:**
     - Caricamento del dizionario nel trie
     - Generazione della matrice di gioco (casuale o da file)
     - Apertura del socket INET e configurazione dei parametri di ascolto
  2. **Gestione delle connessioni:**
     - Accettazione di nuove connessioni e creazione di un thread dedicato per ciascun client
     - Registrazione degli utenti e delle strutture dati (es. lista client, punteggi, bacheca)
  3. **Ciclo Partita/pausa:**
     - **Partita:**

       Inizio della partita con invio della matrice e impostazione del timer di gioco (es. 3 minuti di default). Durante la partita i client inviano le parole che vengono validate e in base al

risultato di validazione (legittimita' della parola proposta, parola non ancora proposta, parola proposta presente sulla matrice) vengono assegnati i punteggi.

- **Pausa:**

  Dopo la fine della partita, viene gestita una pausa (es. 1 minuto) durante la quale vengono accettate richieste limitate (non e' permesso proporre le parole durante la pausa, comando matrice inviato dal client durante la pausa stampa la pausa residua)

4. **Elaborazione dei punteggi e classifica**
   - I thread client inviano i punteggi tramite una coda condivisa
   - il thread scorer, sincronizzato tramite condition variables, elabora e ordina i punteggi, costruendo la classifica in formato CSV, che viene poi mandato a tutti client
5. **Terminazione e pulizia**
   - In risposta a un segnale di shutdown (SIGINT), il server invia un messaggio MSG_SERVER_SHUTDOWN a tutti client, chiude il socket e libera le risorse allocate (inclusi i thread tramite `pthread_cacncel()` e `pthread_join()`

4.2. Client

- **Flusso interattivo**
  1. **Avvio e connessione**
     - Il client si connette al server specificando hostname e porta
     - Viene creato un thread dedicato, tramite la funzione `client_receiver`, che gestisce la ricezione asincrona dei messaggi dal server.
  2. **Registrazione e login**
     - L'utente immette il comando per la registrazione (`registra_utente <nome_utente>`) o per il login (`login_utente <nome_utente>`). per fare il login, deve prima registrare l'utente e il nome con cui vuole fare login non deve essere cancellato tramite `cancella_registrazione <nome_utente>`
     - Il client invia il messaggio appropriato e attende una risposta di conferma (MSG_OK) o un errore (MSG_ERR)
  3. **Interazione con server durante la partita:**
     - Il client richiede la matrice di gioco (comando `matrice`), invia parole tramite il comando `p <parola>` , e puo' interagire con la bacheca tramite comandi `msg <messagio>` e `show-msg`
  4. **Interazione con server durante la pausa:**
     - Come e' stato specificato prima, il client puo' usufruire a tutti i comandi disponibili durante la pausa tranne il comando `p <parola>` visto che la partita non e' in corso.
  5. **Terminazione della sessione**
     - Utilizzando il comando `fine` , il client termina la sessione in maniera ordinata, chiudendo la connessione.

---

5. Struttura del logging

La gestione dei log viene effettuata tramite la funzione `log_event()` .

Ogni evento viene registrato con un timestamp, il nome del server e un messaggio descrittivo, consentendo un'analisi dettagliata delle operazioni eseguite.

Per garantire la coerenza dei dati e prevenire race condition, il logging e' protetto da mutex (`log_mutex`) assicurando l'accesso sequenziale al file di log.

I messaggi di log comprendono:

- **Avvio e Arresto di server:** Registrazione della configurazione iniziale e shutdown del server
- **Connessioni e Disconnessioni**: Accettazione di nuovi client, timeout di inattività e chiusura delle connessioni.
- **Gestione Utenti**: Registrazione, login e cancellazione degli utenti.
- **Invio Parole**: Proposte dei client con esito positivo o negativo e punteggio assegnato.
- **Ciclo Partita**: Avvio, durata, numero di partecipanti e classifica finale.
- **Bacheca Messaggi**: Pubblicazione e visualizzazione dei messaggi.
- **Errori e Debug**: Problemi di comunicazione, parole non valide e anomalie di sistema.

La funzione di logging è progettata per gestire messaggi con un formato variabile, grazie all'uso di argomenti variabili, simile alla funzione `printf`. Inoltre, dopo ogni scrittura, viene eseguito un *flush* del buffer di scrittura, assicurando che i dati vengano effettivamente salvati nel file.

---

6. Struttura e logica dei programmi di test

6.1. Strategie di Test

- **Test Unitari**:
  - **Trie**: Sono stati testati casi di inserimento e ricerca, con particolare attenzione alle parole contenenti "Qu".
  - **Matrice e DFS**: Verifica della correttezza dell'algoritmo DFS su casi di parole contigue in diverse direzioni, inclusi scenari limite in cui si tenta il riutilizzo di una cella.
- **Test di Integrazione**:
  - Verifica della corretta comunicazione tra client e server, controllando il corretto parsing dei messaggi e la gestione degli errori (ad es. timeout, disconnessioni improvvise).
  - Simulazione di più client (fino a 32) per testare la robustezza della gestione concorrente e l'efficacia dei meccanismi di sincronizzazione.

6.2. Strumenti Utilizzati

- **Valgrind**:
  - Impiegato per il controllo dei memory leak e per verificare che non si verifichino accessi illegali alla memoria.
- **Test Manuali**:

- Sono stati condotti test interattivi per verificare il corretto funzionamento dei comandi client, la visualizzazione della classifica e il comportamento in caso di disconnessione o timeout.

6.3. Logica dei Test

- **Scenario 1**: Invio di una parola valida da un client e verifica della risposta corretta (punteggio assegnato).
- **Scenario 2**: Invio di una parola duplicata dallo stesso client per verificare l'assegnazione di 0 punti.
- **Scenario 3**: Test di gestione della disconnessione improvvisa e timeout, verificando la rimozione del client dalla partita.
- **Scenario 4**: Validazione della sincronizzazione tra thread (orchestratore e scorer) tramite condition variables, verificando che la classifica venga comunicata correttamente prima dell'avvio della partita successiva.

---

7. Istruzioni per compilazione ed esecuzione del programma.

7.1. Compilazione

Il progetto e' gestito tramite un Makefile. Per compilare, eseguire sul terminale:

```
make      #compilazione server e client e generazione di eseguibili
make clean # cancellazione gli eseguibili
```

se la compilazione ha successo, verranno generati due eseguibili, `paroliere_srv` e `paroliere_cl`

7.2. Esecuzione server

il Server va avviato specificano il nome del server, la porta e gli eventuali parametri opzionali

```
./paroliere_srv nome_server porta_server [--matrici mtx_path] [--diz dict_path]
        [--durata durata_in_minuti] [--seed int_seed] [--disconnetti-dopo timeout_min]
```

dove:

- `paroliere_srv`: nome del eseguibile
- `nome_server`: nome del server, testato su localhost.
- `porta_server`: numero intero compreso tra 1025 e 65535

opzionali:

- `--matrici` seguito da nome del file da cui caricare matrice (altrimenti generazione casuale)
- `--diz` seguito da nome del file da cui caricare dizionario (altrimenti carica da `resources/dictionary.txt`)
- `--seed` seguito da un intero, il seed da usare per la generazione pseudocasuale di matrice.

- `--durata` seguito da un intero positivo, permette di indicare la durata della partita (default: 3 minuti)
- `--disconnetti-dopo` seguito da un intero positivo, permette di indicare il tempo di timeout per disconnessione (default: 3 minuti)

7.3. Esecuzione Client

il Client va lanciato indicando l'indirizzo del server e la porta

```
./paroliere_cli nome_server porta_server
```

I comandi disponibili per il client includono:

- **registra_utente <nome>**: Per registrare un nuovo utente.
- **login_utente <nome>**: Per effettuare il login (in caso di ripresa della sessione).
- **matrice**: Per richiedere la matrice di gioco e il tempo residuo.
- **p <parola>**: Per inviare una parola al server.
- **msg <testo>**: Per pubblicare un messaggio nella bacheca.
- **show-msg**: Per visualizzare gli ultimi messaggi pubblicati.
- **aiuto:** Per visualizzare la lista di comandi
- **fine**: Per terminare la sessione e disconnettersi.