# 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 e’ suddiviso in due macro-componenti:

- Server:
  - gestisce le connessioni dei client, creando un thread per ogni client connesso.
  - gestisce il ciclo di gioco, compresa la generazione della matrice e il calcolo dei punteggi
  - comunica con i client tramite un protocollo ben definito, inviando messaggi di stato, risultati di gioco e aggiornamenti della matrice.
- Client:
  - fornisce un’interfaccia interattiva per l’utente
  - gestisce la comunicazione con il server
  - visualizza in tempo reale gli aggiornamenti e risultati del gioco

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

---

# 2. Principali scelte del progetto

## 2.1 Strutture dati e algoritmi

- **Matrice di gioco e DFS:**

  La matrice e’ una griglia 4×4 di lettere

  Ci sono tre modi in cui vengono creati le matrici di gioco:

  1. Se viene specificata l’opzione `--matrici` all’avvio del server, la matrice viene caricato dal file specificato tramite la funzione `read_matrix_from_file()`. E ogni volta che si raggiunge EOF viene chiamata la funzione chiamato la funzione `rewind()` e la lettura riprende dall’inizio.
  2. Se viene specificata l’opzione `--seed` all’avvio del server, l’intero passato viene utilizzato come seed per una generazione casuale di matrice tramite la funzione `generate_matrix()`che all’interno di essa utilizza `srand(seed)` e `rand()`.
  3. Nel caso in cui non venisse specificato nessuna delle due opzioni, la matrice viene generata casualmente utilizzando `time(NULL)` come seed.

  L’esistenza della parola proposta dall’utente all’interno della matrice del gioco viene controllata tramite la funzione `is_word_in_matrix()` che all’interno di essa tokenizza la parola (trattando "Qu” come un singolo token) e utilizza un algoritmo di DFS `dfs_find()` che esamina la matrice per verificare se le lettere della parola siano effettivamente collegate tramite celle contigue in tutte le direzioni (orizzontale, verticale e diagonale). Inoltre l’algoritmo implementato, tramite un flag `visited` , assicura che ogni cella venga utilizzata al piu’ una volta.

- **Dizionario delle parole:**

  Per verificare la validita' delle parole, e' stato utilizzato un **Trie,** una struttura dati che consente una ricerca rapida (O(L), L = Lunghezza di parola) . Il dizionario e' caricato all'inizio del gioco tramite la funzione `load_dictionary_trie()` , che legge un file di testo e inserisce le parole nel Trie utilizzando `trie_insert()` . Per gestire il caso speciale della coppia "Qu", viene utilizzato un indice dedicato (26) nell'array dei figli

- **Gestione dei client:**

  I client attivi sono gestiti tramite un array fisso (MAX_NUM_CLIENTS = 32) e ogni client e' rappresentato da una struttura che include il socket, lo stato di connessione, il punteggio e le parole gia' proposte

- **Verifica del nome utente:**

  E' stato implementato la funzione `valida_nome_utente()` da parte di client, che controlla il nome proposto abbia al massimo 10 caratteri e che sia composto solo da caratteri alfanumerici. Invece sul server, il controllo viene ripetuto assicurando che nomi simili (case sensitive) non vengano registrati duplicatamente.

## 2.2. Logica di comunicazione

La logica di comunicazione tra il client e il server e' stata implementata utilizzando **socket stream based** (`AF_INET, SOCK_STREAM`)**.** Ogni messaggio inviato tra client e server e' composto da tre parti:

- **type (1 byte):** Indica il tipo di messaggio, utile per determinare l'azione da intraprendere (MSG_OK, MSG_ERR, MSG_PAROLA, ecc.)
- **length (4 byte)**: Specifica la lunghezza del campo 'data' in **network byte order. L**a funzione `htonl()` e `ntohl()` per garantire una corretta trasmissione dei dati anche tra sistemi con ordine dei byte diverso
- **data (variabile)**: Contiene i dati effettivi del messaggio, come il nome dell'utente o la matrice del gioco, a seconda del tipo di messaggio.

Le funzioni `send_message()` e `receive_message()` che sono responsabili di comunicazione tra client e server, utilizzano i wrapper rispettivamente, `robust_write()` e `robust_read()` che sono state sviluppate per gestire in maniera robusta le operazioni di I/O sui socket, garantendo che l'intera quantita' venga trasferita, anche in presenza di interruzioni (EINTR). rendendo cosi' la comunicazione affidabile.

- `robust_write()`: Questa funzione si assicura che tutti i byte richiesti vengano scritti sul socket. In pratica, se la chiamata a `write()` restituisce un valore inferiore al numero di byte da scrivere (o viene interrotta con un errore EINTR), la funzione ripete la scrittura dal punto in cui era stata interrotta, fino a completare l'operazione.
- `robust_read()` : Analogamente, questa funzione garantisce che venga letto il numero richiesto di byte, ripetendo la lettura se l'operazione viene interrotta da segnali o in caso di parziale lettura

Inoltre, la funzione `send_message()` restituisce **0** se il messaggio viene inviato correttamente, mentre restituisce **-1** se si verifica qualsiasi errore durante l'invio (incluso l'errore EPIPE, che indica che il client ha chiuso la connessione)

### 2.2.1 Impostazione del socket

Durante l'inizializzazione del socket, il server imposta l'opzione `SO_REUSEADDR` tramite `setsockopt()` , la funzione che permette di settare opzione per i socket. questa scelta permette il riutilizzo immediato della porta dopo la chiusura del socket, evitando errori di binding legati ai tempi di attesa.

Ogni client ha un **timeout di inattivita'** configurato tramite la funzione `setsockopt()` , impostando l'opzione `SO_RCVTIMEO` . Se il client non invia dati per un periodo (default 3 minuti, possibile specificare all'avvio del server), la connessione viene chiusa dal server, inviando al client disconnesso il messaggio `MSG_SERVER_SHUTDOWN` per notificare la chiusura

## 2.3 Gestione della concorrenza

Il sistema è progettato per essere multithreaded, creando un thread dedicato per ogni client connesso e altri thread per gestire operazioni centralizzate (es. orchestrator e scorer).

### 2.3.1. Sincronizzazione: Mutex e Condition Variables

- **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 file di log siano mutuamente esclusivi
  - `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.
  - `server_console_mutex` e `client_console_mutex` : proteggono, rispettivamente, l'output sulla console di server e di client, garantendo che i messaggi non vengano mescolati quando piu' thread stampano simultaneamente
- **CV (Condition Variables):**
  - `score_queue_cond` : il thread  Scorer attende che il numero di punteggi raccolti raggiunga il numero atteso
  - `ranking_cond` : come riportato sopra, coordina il flusso tra thread Scorer e Orchestrator

### 2.3.2. Gestione dei thread

**Lato client:**

Il client avvia due thread principali:

- **Main thread:**

  gestisce l'interfaccia utente, legge i comandi da input e coordina l'esecuzione del programma
- **Receiver thread:**

  esegue la funzione `client_receiver()` per ricevere continuatamente i messaggi dal server, garantisce che l'interfaccia rimanga reattiva
  - Riceve aggiornamenti sulla matrice, sui punteggi e altri messaggi di sistema
  - Stampa i messaggi ricevuti e, in caso di errori o shutdown, segnala la terminazione della sessione impostando il flag di terminazione ( `shutdown_flag` )

  Quest'ultimo flag e' dichiarato come `volatile` e `sig_atomic_t` per:

  1. `volatile` : per indicare al compilatore di non cachare la variabile in modo da permettere che le modifiche siano sempre lette dalla memoria.
  2. `sig_atomic_t` : per assicurare operazioni atomiche anche in presenza di segnali, mantenendo l'integrita' del dato

**Lato server:**

il server si avvia con un main thread principale che funge da **dispatcher**. Questo main thread ha il compito di :

- Accettare le nuove connessioni in ingresso tramite il ciclo di `accept()`
- Creare e avviare, per ogni nuovo client connesso, un thread dedicato (tramite `pthread_create()`) per gestire la comunicazione bidirezionale, eseguendo la funzione `client_thread()`
- Avviare thread globali per la gestione di gioco, eseguendo funzione:
  - `orchestrator_thread()` : gestisce il ciclo partita/pausa. All'avvio di ogni partita, rigenera la matrice, resetta lo stato di tutti client connessi (punteggi, parole gia' inviate) e notifica ognuno di essi 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 della classifica cosi' costruita avviene 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

- **Common:**
  - **common.h / common.c:**

    Definiscono funzioni e costanti per la comunicazione (come `send_message()` e `receive_message()`) e le operazioni di I/O robusto (`robust_read()` e `robust_write()` e `robust_write()`) che sono utilizzate sia dal server che dal client.
- **Server:**
  - **server_main.c**:

    Responsabile dell'inizializzazione del server, include il parsing dei parametri della linea di comando (nome, porta, opzioni opzionali), la configurazione del socket (incluso l'uso di `setsockopt()` per abilitare il riuso della porta) e la gestione dei segnali, come SIGINT, per eseguire uno shutdown ordinato.
  - **server_paroliere.c**:

    Implementa la logica centrale del gioco, gestendo:

        - Il ciclo partita/pausa.
        - La creazione e il monitoraggio dei thread dedicati ai client.
        - La sincronizzazione tra thread, ad esempio per l'invio della classifica finale tramite messaggi (MSG_PUNTI_FINALI) e la gestione dei timeout o errori di comunicazione.
    - **dictionary.c**:

      Implementa il Trie per la gestione del dizionario. Include funzioni per:
        - Creare nuovi nodi (`trie_create_node()`).
        - Inserire parole nel Trie (`trie_insert()`).
        - Verificare l'esistenza di una parola (`trie_search()`).
        - Liberare la memoria occupata dal Trie (`trie_free()`).
    - **matrix.c**:

      Gestisce la generazione della matrice di gioco:
        - Generazione casuale con `rand()` e un seed configurabile
        - Lettura della matrice da file tramite `read_matrix_from_file()`.
- **Client:**
    - **client_main.c**:

      Si occupa della connessione al server, inclusa la risoluzione dell'hostname e l'apertura del socket. Avvia inoltre il thread di ricezione dei messaggi.
    - **client_paroliere.c**:

      Fornisce l'interfaccia utente interattiva per l'invio dei comandi. Gestisce un thread separato per ricevere e visualizzare i messaggi provenienti dal server, aggiornando in tempo reale il prompt e permettendo l'invio di parole e altre richieste.
    - **Validazione delle parole**:

      La validazione delle parole proposte dal client viene eseguita tramite un algoritmo DFS, che controlla la componibilità della parola dalla matrice di gioco.

## 3.2. Librerie e API utilizzate

- **Librerie standard C:**

  `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<ctype.h>`, `<unistd.h>`, ecc.
- **API per il multithreading:**

  La libreria POSIX **pthread** (funzioni come `pthread_create()`, `pthread_mutex_lock()`, `pthread_cancel()`, `pthread_cond_wait()`, `pthread_join()` ecc.) usata per gestire i thread e la sincronizzazione.
- **API di rete:**

  Le API dei socket, tramite `<sys/socket.h>`, `<netinet/in.h>`, `<arpa/inet.h>` e l'uso di `setsockopt()` per abilitare il riuso immediato della porta (con SO_REUSEADDR)
- **Gestione dei Segnali:**

  La libreria `<signal.h>` per intercettare segnali come SIGINT e SIGALRM, permettendo uno shutdown ordinato e l'interruzione controllata dei thread.

---

# 4. Struttura dei programmi sviluppati

## 4.1. Server

1. **Avvio e inizializzazione**

   In questa fase vengono eseguiti:

   - **Parsing dei parametri CLI**: il server legge il nome, la porta e le opzioni (come durata della partita, seed, file per matrice/dizionario)

     *Nota*: poiche' l'unico uso di seed all'interno del progetto e' generazione casuale di matrici, e' stato implementato un controllo in cui, all'avvio del server, se vengono specificati sia seed che il path per matrici (opzioni `--seed` e `--matrici`, il sistema restituisce un errore e terminare l'esecuzione in modo da evitare possibile ambiguita' nella generazione della matrice

   - **Configurazione del socket**: viene creato un socket INET in modalita' stream (TCP). L'opzione `SO_REUSEADDR` e' impostata tramite `setsockopt()` per consentire il riutilizzo immediato della porta in caso di riavvio del server.

   - **Gestione dei segnali**: viene impostato un gestore per SIGINT (tramite `sigaction()`), che, al ricevimento del segnale, avvia il processo di shutdown ordinato del server.

2. **Gestione delle connessioni**

Una volta configurato il socket, il server entra in un loop di accettazione delle connessioni:

- Per ogni nuova connessione, il server cerca uno slot libero all'interno dell'array dei client (con numero massimo definito, nel nostro caso 32)
- Per ogni client connesso, viene creato un thread dedicato che esegue la funzione `client_thread()` Questa funzione si occupa di gestire la comunicazione bidirezionale con il client, ricevendo i comandi e inviano le risposte appropriate

3. **Ciclo partita/pausa**

   La logica centrale di gioco e' implementata in `server_paroliere.c`

   - **Ciclo partita:** al momento dell'avvio di una partita, il server genera o carica una matrice di gioco e invia ai client la matrice tramite il messaggio `MSG_MATRICE` insieme al tempo residuo (`MSG_TEMPO_PARTITA`). Durante la partita, i client inviano le parole e il server verifica la loro validita'.

     *Nota*: il server invia immediatamente la matrice e il tempo residuo a ciascun client loggato al nuovo inizio partita

   - **Ciclo pausa:** al termine della partita. il server attende una pausa ( 1 minuto) durante il quale in gioco non viene avviato, ma il server risponde a richieste come la visualizzazione della bacheca o la richiesta di tempo d'attesa (`MSG_TEMPO_ATTESA`) tramite il comando matrice

4. **Elaborazione dei punteggi e classifica**

   - I thread dei client inviano i punteggi in una coda condivisa protetta da opportune primitive (mutex, CV)
   - il thread `scorer_thread` raccoglie, ordina (bubble sort) e costruisce la classifica in formato CSV, che include il vincitore
   - La classifica viene inviata solo ai client che hanno partecipato attivamente alla partita. Il meccanismo di invio forzato dei punteggi considera esclusivamente i client che si sono autenticati

5. **Terminazione e pulizia**

   Il server gestisce lo shutdown in risposta a SIGINT attraverso la funzione `server_shutdown()` :

   - invia le notifiche di shutdown ai client (tramite il messaggio `MSG_SERVER_SHUTDOWN`)
   - Usa `pthread_cancel()` e `pthread_join()` per terminare i thread in modo ordinato e prevenire thread zombie.

- inoltre, libera le risorse ( `trie_free()` , `close()` sui socket, `fclose()` sui file di matrice/log e infine distruzione dei vari mutex e CV)

## 4.2 Client

1. **Avvio connessione**
   Il client si connette al server specificando hostname e porta, risolvendo il nome tramite funzioni di rete (es.
   `gethostbyname()` ). Una volta stabilita la connessione, 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 registrarsi ( `registra_utente <nome_utente>` ) oppure per effettuare il login ( `login_utente <nome_utente>` ).

   - Per il login, è necessario che l'utente sia precedentemente registrato e che il nome non sia stato cancellato tramite `cancella_registrazione <nome_utente>` .
   - **Restrizione:** Fino a quando l'utente non effettua correttamente la registrazione o il login, sono ammessi solo i comandi di registrazione, login, aiuto e fine. Una volta loggato, il server invia immediatamente al client la matrice corrente e il tempo residuo (o il tempo mancante all'inizio della nuova partita), garantendo così che ogni client loggato riceva la notifica e la configurazione di gioco necessaria.
   - Una volta effettuato il login, il server invia immediatamente la matrice e il tempo residuo oppure, in caso di pausa, il tempo mancante all'inizio della nuova partita.
3. **Interazione con il server durante la partita**

   Durante la partita, il client esegue le seguenti operazioni:

   - Richiede la matrice di gioco con il comando `matrice` , ricevendo la griglia e il tempo residuo ( `MSG_MATRICE` e `MSG_TEMPO_PARTITA` ).
   - Invia parole al server tramite il comando `p <parola>` , ricevendo il punteggio associato (con `MSG_PUNTI_PAROLA` ).
   - Può interagire con la bacheca per inviare messaggi (comando `msg <messaggio>` ) o visualizzarla (comando `show-msg` ).
4. **Interazione con il server durante la pausa**
   - Durante la pausa, il client può utilizzare tutti i comandi disponibili, tranne l'invio di parole (comando `p <parola>` ) poiché la partita non è in corso. Vengono, ad esempio, utilizzati i comandi `msg` , `show-msg` e `matrice` (per visualizzare il tempo d'attesa).
5. **Terminazione della sessione**

   Utilizzando il comando `fine` , il client termina la sessione in maniera ordinata:

   - Viene chiusa la connessione con il server.
   - Il thread di ricezione si interrompe e le risorse vengono liberate, garantendo così una chiusura pulita della sessione.

---

# 5. Strutture del logging

La gestione dei log e' centralizzata tramite la funzione `log_event()` , che registra ogni evento con:

- timestamp

- nome del server
- messaggio

ogni scrittura nel file di log viene eseguita in modo sequenziale, grazie all'usao del mutex `log_mutex` , che protegge l'accesso al file e previene race condition

I messaggi di log comprendono:

- **Avvio e arresto di server:** configurazione iniziale, startup e shutdown ordinato
- **Connessioni e disconnessioni:** accettazione di nuovi client, gestione dei timeout e chiusura delle connessioni
- **Gestione Utenti**: operazioni di registrazioni, login e cancellazione degli utenti
- **Invio parole:** proposte inviate dai client, esiti (positivi o negativi) e punteggi assegnati
- **Ciclo partita:** avvio e durata della partita, numero dei 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 `log_event()` accetta argomenti variabili, similmente a `printf()` , e esegue il flush del buffer dopo ogni scrittura per garantire che i dati vengano immediatamente salvati sul file di log.

---

# 6. Struttura e logica dei programmi di test

## 6.1. Strategie di Test

- **Unit Testing:** Verifica delle singole funzioni (es. `send_message()` , `is_word_in_matrix()` , `trie_search()` ) per assicurare la correttezza degli algoritmi e delle strutture dati.
- **Integration Testing:** Simulazione dell'interazione tra i moduli (client-server, gestione del ciclo partita/pausa, registrazione/login) per verificare il comportamento complessivo del sistema.
- 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:

- **Makefile:** Per automatizzare la compilazione dei test insieme al codice principale.
- **Valgrind:** Per il controllo di memory leak e problemi di gestione della memoria.
- **Debugger (gdb):** Per identificare e risolvere eventuali errori di esecuzione e logica.
- **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

*Nota*: per la corretta simulazione si consiglia di specificare l'opzione `--matici resources/matrix.txt` all'avvio del server, dove e' presente la matrice d'esempio presente nel testo di progetto, dove parole legittime sono: CASI, CASE, BEVO, BEVI, VOTA,VOTI, VOLA

- **Scenario 1:**

  Invio di una parola valida da parte di un client e verifica che il server assegni il punteggio corretto. Questo test controlla il funzionamento della validazione della parola, l'accesso al dizionario (Trie) e la corretta elaborazione della matrice di gioco.

- **Scenario 2:**

  Invio di una parola duplicata dallo stesso client per verificare che il server assegni 0 punti, assicurando così che le parole ripetute non influiscano sul punteggio.

- **Scenario 3:**

  Simulazione di una disconnessione improvvisa o di un timeout da parte di un client. Questo test verifica che il server rimuova correttamente il client dalla partita, liberando le risorse associate e aggiornando lo stato del gioco.

- **Scenario 4:**

  Validazione della sincronizzazione tra thread, in particolare tra l'orchestratore e lo scorer, tramite variabili di condizione. Il test controlla che la classifica finale venga comunicata correttamente a tutti i client prima dell'avvio della partita successiva, garantendo una gestione coordinata degli eventi di fine partita.

---

# 7. Istruzioni per compilazione ed esecuzione del programma

## 7.1. Compilazione

Il progetto e' compilabile tramite makefile

```
#compilazione server e client e generazione eseguibili
make

# cancellazione eseguibili
make clean
```

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

## 7.2. Esecuzione server

Il server va avviato specificando 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]
```

- Il parametro `nome_server` è usato come etichetta per il server (ad es. "MyServer", testato con localhost), mentre `porta_server` è la porta su cui il server ascolta (intero compreso tra 1025 e 65535).
- I parametri opzionali permettono di specificare file esterni per la matrice o il dizionario, la durata della partita, il seed per la generazione casuale e il timeout per la disconnessione dei client.
- *Nota:* Se vengono specificati contemporaneamente sia il seed che il percorso per le matrici, il sistema restituisce un errore e termina l'esecuzione, per evitare ambiguità nella generazione della matrice.

## 7.3. Esecuzione client

Il client va lanciato indicando l'indirizzo del server la porta

```
./paroliere_cli nome_server porta_server
```

Una volta connesso, il client avvia una sessione interattiva in cui l'utente può immettere i comandi.

I comandi disponibili per il client includono:

- **registra_utente <nome>**: Per registrare un nuovo utente.
- **login_utente <nome>:** Per effettuare il login
- **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