

RELAZIONE PROGETTO LABORATORIO DI RETI A.A. 2022/23

Progetto Laboratorio di reti: Wordle

Studente: Luca Borrello

Matricola 579438

Laboratorio di reti e calcolatori, Anno Accademico: 2022/2023

I Semestre

INDICE:

1. INTRODUZIONE AL GIOCO
2. SCHEMA GENERALE CLASSI
 - 2.1 PROCESSI
3. STRUTTURE DATI UTILIZZATE
4. COMPILAZIONE ESECUZIONE E MANUALE D'USO
 - 4.1 COMPILAZIONE ED ESECUZIONE
 - 4.2 MANUALE D'USO
5. FUNZIONALITA' DI BASE
 - 5.1 Protocollo comunicazione
 - 5.1 WordleServerMain
 - 5.2 ThreadServer
 - 5.3 WordleClientMain
 - 5.4 Database
 - 5.5 ThreadMultiCast
 - 5.6 Giocatore

1. INTRODUZIONE:

Il progetto consiste nell'implementazione del gioco Wordle, un gioco web-based che consente ai giocatori di provare ad indovinare una parola giornaliera di 5 lettere in 6 tentativi utili ricevendo feedback sui tentativi effettuati, ma con alcune modifiche che rendono l'esperienza di gioco ancora più longeva. Il gioco offre inoltre un aspetto social in quanto permette di condividere i propri suggerimenti dell'ultimo round giocato su una piattaforma e di riceverli su richiesta in qualsiasi momento da altri giocatori connessi. Tra le modifiche troviamo: 12 tentativi utili anziché i classici 6, parole di lunghezza 10 lettere e non 5, il timer di selezione della parola non giornaliero come nella versione originale ma a libera scelta del programmatore.

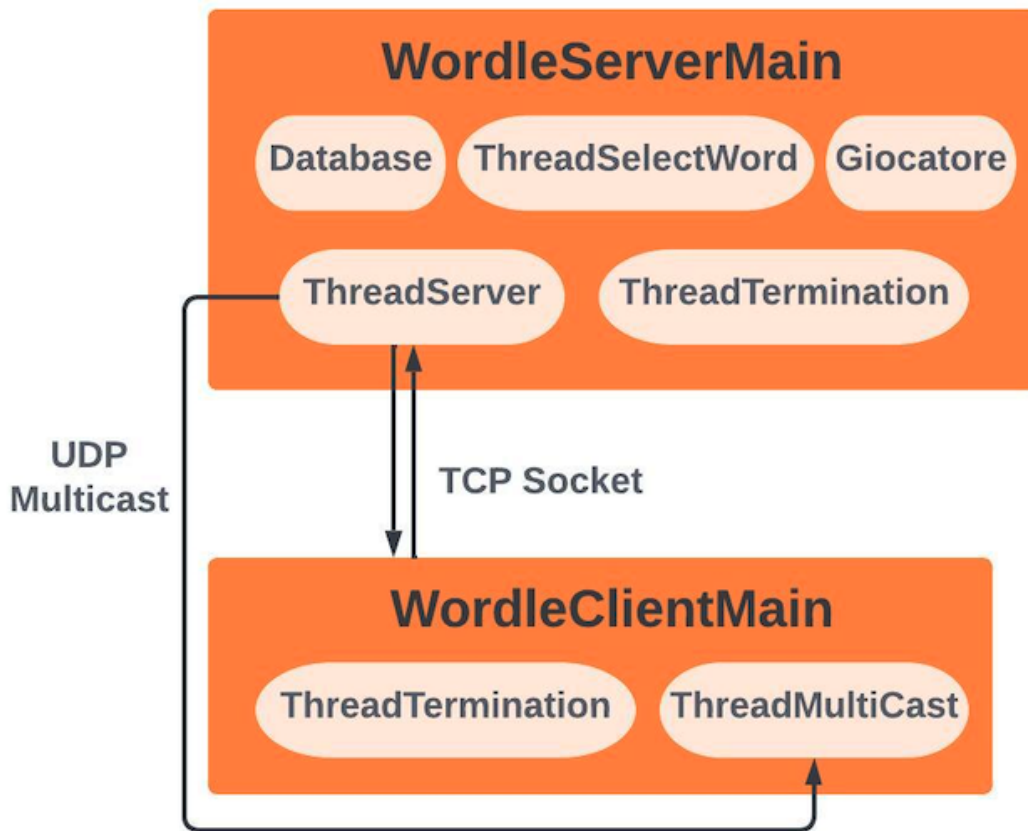
OSSERVAZIONI:

- La mia implementazioni offre delle funzionalità in più non richieste col tracciamento e mantenimento delle parole giocate da parte dei giocatori e quindi alcune strutture dati superflue rispetto a quanto richiesto dal progetto.
- La mia implementazione del progetto permette con un terminale di effettuare nel tempo più accessi a più account senza dover eseguire necessariamente più volte il processo WordleClientMain.java. Infatti eseguendo l'operazione di logout nella schermata principale del gioco, il server non chiude la connessione col client ma semplicemente il giocatore viene disconnesso e solo successivamente ho l'occasione di eseguire nuovamente l'operazione di registrazione e/o logout.
- Con la mia implementazione le statistiche di gioco nel json vengono aggiornate solo quando il server viene spento o per malfunzionamento. Nel mio caso viene simulato quando riceve un segnale di terminazione tramite il comando ctrl + c. Avevo anche implementato altre soluzioni simili che aggiornavano il json in diverse situazioni: subito dopo la disconnessione da parte di un client.
- Ho consegnato il progetto con un file Makefile che semplifica la compilazione. Vedi dettagli nella sezione 5.

2. SCHEMA GENERALE CLASSI:

Viene qui spiegato il funzionamento dei thread e processi principali, dove e quando vengono attivati.

2.1 PROCESSI:



- **LATO SERVER:**

- processo **WordleServerMain**. Si occupa di inizializzare il server e mettersi in attesa di accettare nuove connessioni socket con i client. Parte subito con l'esecuzione del comando.
 - troviamo qui un **CachedThreadPool** di processi **ThreadServer** di default. Utilizzare un cached pool in una situazione come questa si rivela essere la scelta migliore e più efficiente. Una situazione in cui abbiamo un task (client) esattamente per ogni thread che si crea.
 - sempre nel **WordleServerMain** faccio partire un timer per la selezione della parola nuova che altro non è che un thread che aspetta di essere schedulato secondo l'intervallo di tempo che il programmatore sceglie definito nella variabile **PERIOD**, questa viene definita in Secondi nel file config. Terminato il timer il task viene eseguito chiamando il metodo **run()**.
 - ancora nel **WordleServerMain** ho aggiunto uno **shutdownhook**, altro non è che creare un nuovo **ThreadTermination** che permette al server di ricevere il segnale **ctrl + c** ed effettuare una chiusura controllata del server, terminando il pool di thread e aggiornando il file json. Ho deciso di scriverlo così ma potevo creare un'altra classe thread e passarla dentro come parametro al **addShutdownHook()** (classe thread).

- processo ThreadServer. Viene creato quando il server accetta una nuova connessione e dopodiché parte. Questo si occupa di gestire ed elaborare le richieste del client
- LATO CLIENT:
 - processo WordleClientMain. Si occupa di impostare la connessione iniziale col server e dopodiché di gestire l'interazione principale con l'utente.
 - processo ThreadMultiCast . Ogni volta che un utente accede con successo all'account il client crea e fa partire un ThreadMultiCast per il gruppo sociale. Questo ha il compito di ricevere in maniera passiva i messaggi dal server e aggiungerli nella lista.
 - anche qui ho implementato un ThreadTermination per ricevere il segnale ctrl + c ed effettuare una disconnessione controllata del giocatore, mandare i dovuti messaggi al server e chiudere

3. STRUTTURE DATI UTILIZZATE:

Vengono qui spiegate le motivazioni dell'uso delle principali strutture dati. Ho deciso di utilizzare diverse strutture per mantenere le informazioni, tra cui:

1. GAME_ID: utilizzo un AtomicInteger per assicurarmi che la lettura e scrittura sia atomica e non avvengano episodi di concorrenza;
2. SECRET_WORDS: utilizzo una lista di elementi di tipo string, sincronizzata grazie alla libreria java "Collections.synchronized(new Object)". Viene utilizzata per salvare tutte le parole estratte durante una sessione in cui il server è online e viene utilizzata per recuperare e mantenere la SECRET_WORD a cui sta giocando il giocatore, se vengono estratte altre nuove nel mentre. Viene creata nel server main e passata ai ThreadServer.
3. STATS_GIOCATORI: uso una lista sincronizzata come SECRET_WORDS di elementi di tipo Giocatore per mantenere tutte le informazioni sulle statistiche. Ad ogni round terminato questa viene aggiornata da ogni ThreadServer per questo è necessario utilizzare una struttura sincronizzata. E' utilizzata per aggiornare le informazioni nel json. Viene creata nella classe Database e successivamente raccolte le informazioni nel server main.
4. UTENTI_REGISTRATI: ho utilizzato una ConcurrentHashMap che garantisce la mutua esclusione tra thread. Una mappa <username, password> che mantiene le informazioni dei giocatori registrati per rendere più immediato il controllo durante la fase di registrazione. Viene creata nella classe Database e successivamente raccolte le informazioni nel WordleServerMain.
5. UTENTI_CONNESSI: come le precedenti si tratta di una lista sincronizzata di elementi di tipo string che utilizzo per gestire la fase di accesso e mantenere consistente gli utenti che accedono e che si disconnettono. Caso di più utenti che vogliono accedere su più terminali contemporaneamente. Viene creata nel server main e passata ai ThreadServer.

6. LIST_MESSAGE: anche questa una lista sincronizzata (eventualmente se dovessero capitare situazioni in cui il thread multicast sta aggiungendo in coda e il client sta leggendo dalla lista) di elementi di tipo string che viene creata nel processo WordleClientMain e viene passata come parametro al ThreadMultiCast che si occupa di ricevere i messaggi dal server e salvarli proprio in questa struttura dati. Grazie a ciò ogni può recuperare i messaggi da questa struttura dati e non fare richiesta al server.

4. COMPILAZIONE ESECUZIONE E MANUALE D'USO:

Il progetto è suddiviso in cartelle.

1. Config: contiene i due file Properties che vengono utilizzati per la configurazione del client e server.
2. DB: contiene il file json che simula il comportamento di un database in quanto crea le informazioni che vengono salvate nelle strutture dati.
3. Jar: contiene i due file Jar per client e server come richiesto dalle specifiche del progetto. Permette anche un metodo alternativo di esecuzione del progetto.
4. Src: cartella destinata ai file sorgenti ovvero i file .java e post compilazione .class che compongono il progetto
5. Text: cartella destinata al file testuale contenente tutte le parole che possono essere estratte.

Sempre nel progetto è presente un file Makefile che ho creato per semplificare la sintassi per la compilazione ed esecuzione dei file .java e .class.

4.1 Compilazione ed esecuzione (MacOS):

Di seguito viene riportato come eseguire con i comandi la compilazione ed esecuzione. Tutte le operazioni e comandi devono essere eseguiti nella cartella del progetto WORDLE_PROJECT.

- COMPILAZIONE:
 - `javac ./Src/WordleClientMain.java` (LATO CLIENT)
 - `javac -cp ./Lib/gson-2.10.jar ./src/WordleServerMain.java` (LATO SERVER)
- ESECUZIONE:
 - `java Src/WordleClientMain` (LATO CLIENT)
 - `java -cp ./Lib/gson-2.10.jar Src/WordleServerMain` (LATO SERVER)

- ESECUZIONE ALTERNATIVA CON .JAR:
 - `java -jar ./Jar/WORDLE_CLIENT.jar` (LATO CLIENT)
 - `java -jar ./Jar/WORDLE_SERVER.jar` (LATO SERVER)

Sempre dentro la cartella WORDLE_PROJECT si ha la possibilità di eseguire dei comandi precisi alternativi ai classici grazie all'utilizzo del Makefile:

- COMANDI CON MAKEFILE:
 - `make`: esegue i due comandi per la compilazione del client e server;
 - `make class`: rimuove tutti gli eseguibili .class nella cartella Src del progetto;
 - `make c`: comando per eseguire il client;
 - `make s`: comando per eseguire il server;
 - `make c-jar`: comando per eseguire client con file .jar;
 - `make s-jar`: comando per eseguire server con file .jar;

4.2 MANUALE D'USO:

Vengono qui spiegati i comandi principali, come utilizzarli durante la sessione e soprattutto il modo in cui mostro le notifiche dei giocatori.

All'avvio del processo client con successo viene chiesto fin da subito una scelta:

1. [REGISTRAZIONE]
2. [ACCESSO]
3. [CHIUDI TERMINALE]

Nel progetto viene consegnato un file .json simulando un database contenente le informazioni dei giocatori. Questo file contiene già 2 giocatori che possono essere utilizzati per il testing. In alternativa è possibile creare altri nuovi utenti.

Tutte possono essere selezionate sia inserendo il comando (maiuscolo o minuscolo) sia inserendo il numero del comando.

Sia Registrazione che Accesso richiedono Nome utente e Password in due scansioni differenti. Chiudi terminale termina la sessione e chiude la console.

Una volta eseguito l'accesso ci vengono stampati le possibili scelte da fare:

1. [GIOCA]: Inizia un nuovo round di wordle;
2. [STATISTICHE]: Mostra le statistiche di gioco dell'utente;
3. [CONDIVIDI]: Condividi i risultati del gioco sul social;
4. [NOTIFICHE]: Mostra notifiche di gioco inviate dai giocatori;
5. [ESCI]: Termina la sessione di gioco
6. [HELP PER LEGENDA].

Anche qui tutte le azioni possono essere selezionate sia inserendo il comando (maiuscolo o minuscolo) sia inserendo il numero del comando.

Quando la richiesta di gioco va a buon fine, ho due possibilità di azioni:

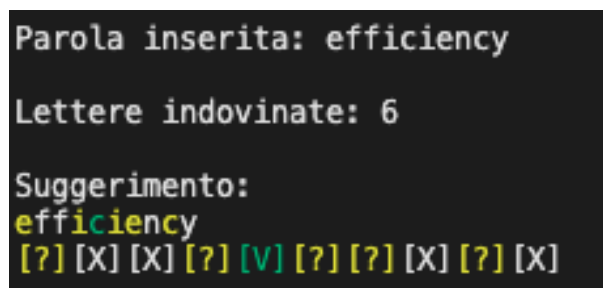
1. [GUESSED WORD]: Inserisci la tua guessed word;
2. [ESCI]: Termina la sessione di gioco.

Quando inserisco una parola i colori indicano lo stato della lettera:

- Presente nella SECRET_WORD e in posizione corretta (colore verde - “[V]”) (Ho cambiato dalle specifiche che richiedeva “[+]”)
- Presente nella SECRET_WORD ma in posizione errata (colore giallo - “[?]”)
- Assente nella SECRET_WORD (nessun colore/grigio - “[X]”)

Il suggerimento mi viene mostrato in due modi per facilitare la visibilità degli errori.

Ad esempio:



```
Parola inserita: efficiency
Lettere indovinate: 6
Suggerimento:
efficiency
[?] [X] [X] [?] [V] [?] [?] [X] [?] [X]
```

Il secondo modo di visualizzare il messaggio è quello che compone il suggerimento che sarà inviato al server e visto dagli altri utenti.

Ad esempio:


```

Ricevuto pacchetti:

Messaggio 1:
Nome utente: admin – Round Wordle 16: 5/12

[?] [?] [?] [X] [X] [?] [X] [?] [X] [X]
[?] [?] [X] [X] [X] [V] [X] [?] [X] [V]
[?] [X] [?] [?] [?] [X] [X] [?] [X] [X]
[?] [X] [X] [?] [V] [?] [?] [X] [?] [X]
[V] [V] [V] [V] [V] [V] [V] [V] [V] [V]

```

Per smettere di giocare se l'utente sta cercando di indovinare la parola può selezionare l'opzione “esci” e successivamente verrà portato alla schermata principale. Dopodiché l'utente può proseguire chiedendo di uscire sempre col comando “esci”. A questo punto un altro utente può accedere o registrare, o se si vuole terminare l'esecuzione del programma si digita il comando “chiudi terminale”.

5. FUNZIONALITA' DI BASE:

5.1 Protocolli comunicazione:

1. TCP: ho utilizzato il protocollo standard di comunicazione TCP che prevede sia instaurata una connessione per poter instradare i messaggi attraverso uno stream socket. Questo tipo di comunicazione è quella prevista nella completa comunicazione e scambio di richieste tra processo il WordleClientMain e il processo ThreadServer che elabora le richieste. Viene aperto un server socket con la porta scelta dal programmatore lato server e una socket lato client con porta e indirizzo scelto dal programmatore. Successivamente ogni volta che il processo WordleServerMain accetta una connessione da parte di un client, questa connessione socket viene utilizzata e mandata al ThreadServer per permettere la comunicazione.
2. UDP: ho utilizzato questo tipo di comunicazione che prevede lo scambio di pacchetti (come dice il nome: user datagram packet) unicamente per far comunicare ogni singolo ThreadServer con il suo relativo processo ThreadMultiCast il quale aspetta di ricevere messaggi . Lato client troviamo sempre la definizione di un socket multicast e la partecipazione da parte di quest'ultimo ad un gruppo sociale grazie all'indirizzo scelto dal programmatore. Il client si occupa di ricevere i pacchetti quindi viene chiamato “receiver”. Lato server funziona in egual modo ma quest'ultimo inviando i pacchetti viene chiamato “sender”. Successivamente il WordleServerMain passa ai ThreadServer tutte queste informazioni per poter comunicare e mandare il pacchetto attraverso il socket multicast. In egual modo il ThreadMultiCast ha la possibilità di ricevere nuovi messaggi da questo socket multicast che viene instaurato e salvare il messaggio nella struttura dati.

5.2 WordleServerMain:

WordleServerMain.java è il processo principale lato server che si occupa di gestire e accettare nuove connessioni con diversi clienti tramite socket, tenere traccia delle informazioni ed elaborare le richieste ricevute dai diversi client e soprattutto chiamare la funzione per aggiornare il json una volta terminato il processo. Imposta tutto il necessario per far funzionare il server, database, connessioni multicast, lettura parametri, quindi inizia ad accettare nuove connessioni.

OSSERVAZIONI:

1. Quando parte: quando viene compilato il programma tramite comando senza nessun fallimento
2. Quando termina: quando riceve un segnale di terminazione brute force (ctrl + c) viene catturata l'eccezione e termina in maniera controllata chiudendo ciò che deve essere chiuso
3. Gestione aggiornamento json:
 - a. avevo in mente diverse soluzioni per la situazione di aggiornamento del json. Ogni volta che un giocatore indovina la parola dopo aver aggiornato le statistiche nelle strutture dati, solamente quando il terminale client termina e quindi dopo richiesta del giocatore, solo quando il server termina per un malfunzionamento o periodicamente ogni volta che viene estratta una parola nuova.
 - b. La versione più efficiente sarebbe quella di aggiornare unicamente il json in caso di terminazione del server poiché i processi dopo la creazione del database lavorano sulle strutture dati sincronizzate. Ho deciso di proseguire in questo modo e aggiornare il json solo in caso di terminazione (simulata quando il server main riceve il segnale ctrl + c e viene chiamata la funzione `updateJson()`)

Ho implementato i seguenti metodi:

- `Main()`:
 - fin da subito chiama `readConfig` per leggere i parametri. Avvia il processo `WordleServerMain` dove viene richiesta una connessione socket con indirizzo specificato
 - successivamente imposta tutte il necessario per il funzionamento del server, database, la selezione della parola, impostazioni server, impostazioni multicast, fa partire il server che accetta infinite connessioni con i client , se termina per malfunzionamento chiude la connessione multicast.
- `ReadConfig()`:
 - metodo che apre il file con `Server.Properties` contenente tutte le informazioni essenziali per i parametri da passare alle varie funzioni. Viene caricato il nome del file delle parole, il nome del file json che simula il nostro database contenente tutte le informazioni dei giocatori persistenti anche dopo lo spegnimento del server, la porta del socket, l'indirizzo e porta del gruppo multicast,

il periodo di aggiornamento della parola da scegliere (importante, in secondi) e il timeout da aspettare perché possano terminare i thread del pool, dopodiché si spegnerà il server.

- `StartSelectWord()`:
 - in questa funzione utilizzo un Timer con uno specifico task che io ho dichiarato. Schedulo il task con una certa frequenza che ho definito nel period per scegliere la parola. Il task viene eseguito come termina il periodo di timer scelto.
 - metodo che imposta con un periodo fissato dal programmatore una parola scegliendo in maniera casuale una riga per volta fino ad un massimo del valore del numero di righe totali del file ricavate calcolando sapendo la lunghezza totale del file e che le parole sono ugualmente lunghe. Una volta scelta la parola viene incrementato il valore del `GAME_ID` e salvata la parola nella variabile `SECRET_WORD` e aggiunta alla struttura dati sincronizzata `SECRET_WORDS` per ricordare tutte le parole giocate.
 - successivamente il task sarà schedulato con il periodo fissato scegliendo una parola nuova per il round di Wordle. E' ammesso che una parola possa essere estratta più volte e anche se irrealistico, consecutivamente.
- `SetupServer()`:
 - salva nelle strutture dati locali le informazioni degli utenti come le statistiche dalla classe Database che ha la funzione di servire da vero e proprio database quando il server viene avviato.
 - OSSERVAZIONE: E' da tenere in considerazione le strutture dati che ho utilizzato per le informazioni e statistiche dei giocatori:
 - `STATS_GIOCATORI`: forse la struttura dati più importante, una lista sincronizzata contenente oggetti di tipo `Giocatore`, che utilizzo per mantenere consistente le informazioni dei giocatori, l'aggiornamento dei dati e aggiornamento del json nel processo `ThreadServer`;
 - `UTENTI_REGISTRATI`: una `ConcurrentHashMap` coppia `<String, String>` che mi garantisce il controllo atomico per tenere traccia degli utenti registrati e controllare quindi il username;
- `SetupMultiCast()`:
 - salva nelle strutture dati locali le informazioni riguardanti il gruppo multicast come il socket, l'indirizzo e si unisce con queste informazioni al gruppo social.
- `CloseMultiCast()`:
 - controlla se il socket del gruppo social esiste e non è chiuso ed eventualmente lascia il gruppo social multicast e chiude il socket.
- `StartServer()`:

- il cuore del processo server, un loop infinito in attesa di potenziali infiniti client che desiderano giocare. Il processo si blocca nella chiamata accept finché un client non si connette creando il socket e dopodiché viene creato e aggiunto il thread del relativo client al pool thread executors che si occupa di gestire i threads. Ho aggiunto un handler al runtime che permette di ricevere quando il processo termina per una anomalia di una operazione o ad esempio per una chiusura forzata del server col controllo “ctrl + c” in modo da gestire in maniera controllata la chiusura del pool thread.

5.3 ThreadServer:

ThreadServer.java è la componente centrale lato server che si occupa di comunicare tramite protocollo TCP direttamente con il proprio client scambiando messaggi, riceve le richieste delle operazioni, elabora i dati, rimanda al client il risultato della operazione, gestisce la registrazione, accesso e logout del giocatore ed ha il compito di aggiornare le statistiche dei giocatori nelle strutture dati concorrenti.

OSSERVAZIONI:

1. Quando parte: quando viene eseguita la thread.start() nel server main
2. Quando termina:
 - a. quando un giocatore invia il comando “Chiudi terminale” per terminare sia lato client che server-thread
 - b. quando viene catturata una eccezione per malfunzionamento da parte del client (Ctrl + c)
 - c. quando il server main crasha e il threadpool esegue una shutdownnow per terminare tutti i thread del pool

Ho implementato i seguenti metodi:

- CreateCommunication():
 - si occupa di stabilire la connessione tra thread-server e client utilizzando il socket che è stato passato dal processo server main. Questo metodo viene invocato subito dopo il salvataggio delle variabili dal costruttore.
- CloseConnection():
 - si occupa di terminare la connessione. Controllo se il socket è ancora aperto ed eventualmente chiudo i canali di comunicazione e il socket
- Run():

- dove il thread-server parte quando viene invocato il metodo `thread.start()`. Un loop infinito in cui il thread si blocca finché non riceve un messaggio dal client a causa della chiamata bloccante `ReadLine()`. Qui il thread-server gestisce le 3 possibili scelte del giocatore: registrazione, accesso e chiusura terminale con le dovute azioni. Se l'operazione di accesso va a buon fine allora inizia la sessione di gioco.
- **Register():**
 - metodo che controlla se i dati inseriti permettono una registrazione di un nuovo giocatore o se il nome utente è stato già scelto. Se la registrazione va a buon fine viene creato un nuovo giocatore e viene invocata la funzione che aggiorna il json.
 - sfrutto l'operazione atomica `putIfAbsent(String,String)` per gestire in maniera efficienti la possibilità che più giocatori contemporaneamente vogliano registrarsi con lo stesso username.
- **Login():**
 - metodo che controlla se lo username inserito esiste, eventualmente se la password è corretta ed eventualmente se ha già effettuato l'accesso in un'altra sessione, in caso negativo allora esegue l'accesso in alternativa manda errore e rifiuta il login.
- **Logout(boolean indovinata, boolean crash):**
 - con questo unico metodo e più parametri gestisco le diverse possibili situazioni in cui un giocatore possa uscire:
 - **IN GAME:** se il giocatore vuole uscire dal game allora la partita viene abbandonata, il giocatore viene riportato alla schermata di gioco principale e le statistiche vengono aggiornate
 - **NELLA SCHERMATA DEI COMANDI:** se il giocatore decide di uscire nella schermata principale di gioco allora l'utente viene disconnesso e il giocatore viene riportato alla schermata iniziale in cui può scegliere se registrarsi accedere o chiudere il terminale.
 - **PER CRASH:** se avviene una situazione di errore e conseguente crash da parte del giocatore (client) allora avviso il server con un codice di errore che il giocatore è crashato e lo disconnette.
- **StartSession():**
 - il login è stato effettuato con successo e inizia la sessione di gioco, server thread cicla all'infinito in attesa di ricevere una richiesta dal client tra le scelte possibili nel menu. Ho gestito manualmente la fase di logout dell'utente per crash se viene ricevuto il codice errore `CRASH_ERROR_CODE`;
 - se il server thread riceve la richiesta di gioco viene invocato il metodo `checkPlayer` che controlla se il giocatore ha i diritti di giocare il round ed in caso di successo viene chiamata la funzione `StartGame` che porta alla fase successiva del gioco.

- **StartGame():**
 - qui vengono da subito inizializzati i tentativi a disposizione. Dopodiché finché non vengono raggiunti i tentativi totali si scorre un loop, il server thread riceve la parola dal canale di comunicazione ed elabora subito se presente o meno nel file delle parole invocando il metodo `checkWord()`. Se indovinata aggiorna le statistiche e porta l'utente alla schermata principale, se i tentativi sono terminati termino sessione e aggiorno statistiche;
 - anche qui come nel precedente metodo ho gestito manualmente due fasi importanti. Ad ogni iterazione dopo aver ricevuto la parola inserita dal giocatore controllo:
 - se è stata estratta una nuova parola col metodo `checkSecretWord` e in caso di successo mando codice errore `NEW_WORD_ERROR_CODE` al client, termino sessione e aggiorno statistiche;
 - se ho ricevuto dal giocatore il segnale di fermarmi ad indovinare allora termino sessione di gioco per abbandono e aggiorno statistiche.
- **ShowStats():**
 - una delle 5 operazioni che il giocatore può richiedere dal menu principale. Ho implementato 2 metodi perché inizialmente avevo frainteso la richiesta del metodo `ShowMeSharing` credendo di dover visualizzare al giocatore tutte le informazioni delle partite degli altri giocatori. Il metodo standard semplicemente scorre nella struttura dati delle statistiche degli utenti finché non trova il giocatore della sessione in corso, raccoglie tutte le informazioni sotto formato di stringa e le invia al client che le stampa sullo `stdout`.
- **ShareStats():**
 - una delle 5 operazioni che il giocatore può richiedere dal menu principale. Il server thread legge dal client il messaggio che gli sta inviando finché non arriva al segnale di terminazione lettura "EOF", compone il messaggio come stringa e lo passa come parametro al metodo `SendMessageSocial()` che si occupa di inviarlo al socket multi cast.
- **SendMessageSocial(string message):**
 - riceve in input il messaggio composto, converte il messaggio da stringa a byte per prepararlo, crea il pacchetto UDP come `DatagramPacket` conoscendo il gruppo e porta multicast e infine manda il pacchetto al thread multicast tramite il socket.
 - se si verifica un errore viene catturata una eccezione e il sistema viene chiuso.
- **CheckSecretWord():**
 - metodo che permette di controllare se viene estratta una nuova parola durante la sessione di gioco semplicemente confrontando un id di gioco temporaneo che viene inizializzato alla partenza del

server e viene aggiornato proprio in questo metodo ogni volta che il GAME_ID è maggiore di quello temporaneo.

- una volta confrontati gli ID viene aggiornata la current SECRET_WORD dalla struttura dati SECRET_WORDS sincronizzata contenente tutte le parole estratte per sessione e viene aggiornato il TEMP_ID.
- CheckPlayer():
 - permette di controllare nella struttura dati dei giocatori se l'identificatore ultimoidgiocatore è diverso da quello attuale GAME_ID. In caso positivo allora il player può giocare.
 - con LAST_PLAYED_GAME_ID tengo traccia dell'indice della parola che sta cercando di indovinare che è presente nella struttura SECRET_WORDS
 - salvo l'attuale id del game nella variabile del giocatore ultimoidgiocato
 - la parte più importante di questo metodo è impostare il segnale GAME_STARTED = true. Ciò mi permette di tenere traccia dello stato di gioco del giocatore nel momento in cui viene chiamata la funzione logout().
 - infine vengono inviati i diversi messaggi di fallimento o successo con il GAME_ID che servirà per comporre il messaggio
- CheckWord(String word):
 - metodo che elabora i suggerimenti se la parola è valida. Viene invocata una binarySearch() della parola per controllare se la parola è presente o no nel file delle parole. Mando i relativi messaggi di errore o successo.
 - una volta elaborato il suggerimento lo invio al client. Ho deciso di elaborare il suggerimento con i colori.
- BinarySearch(String word):
 - ho deciso di utilizzare una ricerca binaria sul file e quindi effettuare ogni volta un accesso al file per controllare la presenza o meno della parola rifiutando la soluzione alternativa di caricare tutto il file in memoria in una struttura dati. Sfruttando la ricerca binaria ho una complessità in tempo di $O(n \cdot \log(n))$ ma perdo in prestazione dovendo accedere per ogni parola al file per ogni thread (giocatore) che sta cercando di indovinare.
 - sfrutto il fatto che il file è regolare con tutte le parole della stessa lunghezza. Di conseguenza ho due margini left e right, calcolo la metà ogni volta, mi posiziono il puntatore del lettore con la funzione seek() e leggo la parola finché non la trovo o fino ad esaurimento file e ritorno successo o fallimento.
- UpdateStats(boolean indovinata):

- ho tutte le statistiche dei giocatori nelle strutture dati sincronizzate e concorrenti, questo mi permette di modificare direttamente i campi di cui ho bisogno del giocatore. Utilizzando il parametro indovinata posso distinguere i due casi di successo o fallimento nell'indovinare la parola del round e compiere azioni diverse. Elaboro i risultati e modifico gli attributi del giocatore.
- importante notare che come eseguo una updateStats() significa che il round di gioco è terminato e per questo imposto subito GAME_STARTED = false.
- Send(String str):
 - funzione che isola la responsabilità di inviare sullo stream al client un messaggio.
- Receive():
 - isolo la responsabilità di ricevere un messaggio dallo stream del client.
 - se succede un errore o un malfunzionamento lancio un'eccezione che viene catturata dagli altri metodi. Ritorno il messaggio ricevuto dallo stream in caso di successo.
- ps(String str):
 - funzione implementata per comodità di stampare messaggi in maniera sintetica.

5.4 WordleClientMain:

WordleClientMain è il processo centrale lato client. Viene eseguito un processo per ogni giocatore che ha intenzione di collegarsi al server e giocare. Preserva tutta l'interazione tra client e server durante la connessione, manda le richieste e riceve le risposte a quest'ultime dal server.

OSSERVAZIONI:

1. Quando parte: quando viene compilato il programma tramite comando senza nessun fallimento
2. Quando termina:
 - a. quando invio il comando "Chiudi terminale" per terminare la sessione del client
 - b. quando riceve un segnale di terminazione brute force (ctrl + c) viene catturata l'eccezione e termina
 - c. quando il server termina per malfunzionamento viene catturata l'eccezione, il controllo running viene impostato a false e il client termina in maniera controllata senza lasciare nulla aperto

Ho implementato i seguenti metodi:

- Main():

- fin da subito chiama readConfig per leggere i parametri. Avvia il processo WordleClientMain dove viene aperto uno scanner per lo standard input dell'utente e avvia una connessione socket con indirizzo e porta specificati e crea la comunicazione col metodo CreateCommunication().
- fa partire la sessione del client e se termina il processo segue il flusso e termina correttamente.
- ReadConfig():
 - già descritto nel WordleServerMain. Vengono lette le informazioni necessarie che vengono utilizzate come parametri per le funzioni.
- CreateCommunication():
 - già descritto nel ThreadServer
- CloseCommunication():
 - già descritto nel ThreadServer
- SetupMultiCast():
 - già descritto nel WordleServerMain
 - piccola differenza qui nel processo client, per poter attendere in maniera passiva i messaggi senza essere bloccati bisogna attivare un thread che si occupi in maniera indipendente dal processo client della ricezione dei messaggi
 - creo quindi la classe ThreadMultiCast con i dovuti parametri e faccio partire il task del thread con la chiamata start().
- CloseMultiCast()
 - già descritto nel WordleServerMain
- StartClient():
 - metodo che viene invocato dal processo main. Da qui inizia la vera e propria sessione di gioco dell'utente in cui può interagire tramite richieste col server
 - un loop infinito in cui il giocatore rimane finché non esegue l'accesso in maniera corretta. Le opzioni sono registrare un nuovo utente, accedere con le credenziali conosciute o terminare la sessione del client terminando il processo. Con terminazione controllata viene chiuso il gruppo multicast (se effettuato login), la connessione e il processo termina regolarmente
 - se l'accesso viene eseguito con successo viene instaurata la connessione col gruppo social multicast chiamando il metodo SetupMultiCast() e inizia la vera sessione di gioco col menu principale. Importante notare che se eseguo con successo l'accesso la coda dei suggerimenti viene svuotata perché potrei aver effettuato l'accesso con un altro account e vedere comunque gli stessi suggerimenti.

- ho implementato un addShotDownHook (come nel caso già descritto del WordleServerMain) letteralmente un “uncino” che altro non é che un thread che cattura situazioni di malfunzionamento precise:
 - se l'utente simula un malfunzionamento col segnale ctrl + c questo viene catturato, viene mandato il codice di errore CRASH_ERROR_CODE al server e si prepara per la terminazione del processo client, mentre il server si occupa di effettuare una disconnessione regolare, se l'utente aveva effettuato l'accesso e ad aggiornare statistiche e json
- Register():
 - vengono chiesti all'utente username e password, vengono mandati al server insieme ad una richiesta di registrazione. Il server elabora la richiesta, controlla username e password e risponde con un codice
 - il client traduce codice e riferisce all'utente se è andata a buon fine o ci sono stati errori
- Login():
 - come register(), vengono chiesti all'utente username e password, vengono mandati al server insieme ad una richiesta di registrazione. Il server elabora la richiesta, controlla username e password e risponde con un codice
 - il client traduce codice e riferisce all'utente se è andata a buon fine o ci sono stati errori
- Logout():
 - si limita a mandare una richiesta di logout() al server e quest'ultimo si occupa di verificare in che situazione il client si trova, elabora tutte le procedure che servono e disconnette il giocatore dalla sessione
- StartSession():
 - qui il giocatore ha effettuato l'accesso con successo e adesso si prepara a giocare. Vengono stampati su console i principali comandi e il giocatore ha la possibilità di scegliere uno di questi.
 - se decido di giocare si passa alla fase di gioco in cui l'utente può indovinare (se ha il diritto di giocare)
 - OSSERVAZIONE:
 - ho implementato il logout in modo tale che se viene chiamata in questa schermata, il giocatore viene riportato alla schermata di registrazione e non viene terminato il client
- PlayWorlde():
 - viene fatta richiesta di iniziare a giocare, il client manda richiesta al server, quest'ultimo riceve il segnale, controlla se il giocatore può giocare e risponde al client con un codice. Se va a buon fine il giocatore può giocare e iniziano i tentativi senno ha già giocato il turno e deve aspettare

- se tutto va a buon fine, viene composta la prima parte del messaggio che può essere condiviso sul gruppo social. Questo contiene le informazioni del nome utente del giocatore che sta condividendo e il round di wordle che a cui sta giocando. Successivamente viene attivato il metodo `startGuessing()`
- `StartGuessing()`:
 - un loop controllato che scorre per tante volte quanti sono i tentativi a disposizione. Se in questa fase il client riceve dall'utente il messaggio esci, il client informa il server e fa terminare la sessione della parola attuale, vengono aggiornate le statistiche e l'utente viene riportato al menu principale con le operazioni, non viene disconnesso.
 - in alternativa può provare ad indovinare la parola inserendo il comando numerico 1 o “guessed word”
 - una volta terminati i tentativi viene avvisato l'utente e il messaggio per il gruppo social viene composto con i tentativi effettuati dall'utente e i suggerimenti mandati dal server
- `SendWord()`:
 - fase in cui il giocatore può finalmente provare ad indovinare la parola scelta dal server. Se viene inserita una parola più lunga, più corta o non presente nel dizionario il tentativo non viene contato e quindi non incremento TENTATIVO. Al giocatore viene chiesto di inserire la sua parola proposta che viene confrontata con quella segreta dal server
 - client manda parola al server che la confronta e risponde con un suggerimento se presente nel vocabolario. Questo suggerimento viene salvato fino a comporre tutti i suggerimenti ricevuti
 - se ha indovinato tutte e 10 le lettere ritorna successo e il giocatore dovrà aspettare un nuovo round, in alternativa viene stampato il suggerimento ricevuto dal server
 - è possibile che durante questa fase venga estratta una nuova parola, se ciò accade il server avverte con un messaggio il client e la sessione viene terminata aggiornando le statistiche come sconfitta da parte del giocatore
- `SendMeStatistic()`:
 - con questo comando l'utente può richiedere al server le sue statistiche aggiornate all'ultima partita. Il client manda segnale di richiesta, server riceve messaggio, elabora dati e manda al client, quindi stampa su console i risultati
 - ho utilizzato un segnale di terminazione lettura “EOF” per sapere quando fermarmi
- `Share()`:
 - con questo comando l'utente può richiedere al server di condividere i suggerimenti della sua ultima partita effettuata.

- se il messaggio da condividere è vuoto allora il giocatore non ha ancora giocato nessun round, in alternativa il client manda segnale richiesta, messaggio e segnale terminazione “EOF” al server, quest'ultimo invia il messaggio del giocatore al gruppo social
 - OSSERVAZIONE: questo metodo comunica col server per elaborare la richiesta ma se si verifica una situazione di crash del server non viene percepito. Per questo faccio mandare dal server un messaggio di stato “OK_SHARE” se va tutto bene. Se il server termina per malfunzionamento il client cattura l’eccezione e termina.
- ShowMeSharing():
 - con questo comando l'utente può richiedere di ricevere le notifiche che gli altri giocatori hanno inviato al gruppo social riguardo le loro partite effettuate.
 - poiché il client avvia un thread direttamente per occuparsi della ricezione dei messaggi in una lista condivisa di stringhe, il client non fa altro che leggere da questa struttura dati condivisa tutti i messaggi che sono stati condivisi dagli altri giocatori con un loop controllato. Se ci sono i messaggi , in alternativa nessun giocatore ha ancora condiviso
 - OSSERVAZIONE: questo metodo non comunica col server per elaborare la richiesta perciò se si verifica un episodio di crash del server e io sono collegato per come é stato implementato il codice il giocatore potenzialmente può comunque visualizzare le notifiche poiché sono salvate lato client. Solamente chiamando un altro dei metodi allora viene catturata l’eccezione e fatto terminare il client.
- Receive():
 - già descritto nel ThreadServer
- Send(String str):
 - già descritto nel ThreadServer
- ps():
 - già descritto nel ThreadServer
- ReceiveInput():
 - metodo creato per comodità. Permette di leggere messaggi da tastiera dall'utente

5.5 Database:

Ho creato una classe database che svolge la funzione di raccolta dati, effettua l'operazione di popolare le strutture dati dei giocatori e delle statistiche dal file json persistente contenente tutte le informazioni utili e infine di aggiornare il file json una volta che il server termina.

La classe riceve come parametri solo il nome del file da Database json da aprire e l'identificativo atomic volatile della sessione di gioco. Qui nella classe inizializzo anche le strutture dati più importanti che verranno utilizzate nel programma per mantenere consistenza dei dati. Sono presenti i seguenti metodi:

- **SetUpDatabase():**
 - inizia col creare le due variabili quello json e successivamente il reader per leggere dal file json. Se il database alla prima inizializzazione risulta vuoto, viene creato con la prima registrazione andata a buon fine
 - apre in lettura il file json utilizzando e sfruttando le funzioni che vengono importate dalla libreria gson di google. Si leggono tutte le informazioni di un giocatore, si crea il nuovo giocatore e si salva nelle strutture dati appropriate.
- **UpdateJson():**
 - ho deciso di effettuare il salvataggio delle statistiche nel json al momento della chiusura o in caso di eventuale terminazione improvvisa del sistema lato.
 - nel mio caso di progettazione quindi i dati vengono salvati sul json solo al momento dell'invio del segnale ctrl + c al server
 - utilizzando la libreria di google per la serializzazione gson creo un nuovo json con le informazioni aggiornate delle statistiche dei giocatori
 - per ogni giocatore nella struttura dati converto il formato in un json giocatore con la funzione toJson().

5.6 ThreadMultiCast:

Ho creato una classe thread separata che si occupa di gestire i messaggi datagram packet (UDP) che vengono inviati dal server. Vengono passati al processo il socket per il multicast, la coda sincronizzata per i suggerimenti e la lunghezza del buffer.

- **Run():**
 - Quando viene fatto partire inizia un loop infinito in attesa, sulla funzione bloccante receive(), dei server che mandano il messaggio. Una volta arrivato un nuovo messaggio UDP questo viene prima convertito e poi salvato nella struttura dati dedicata così che gli altri client possano leggere e stampare i suggerimenti
- **ReceiveMessage():**
 - crea un nuovo pacchetto UDP con la lunghezza del buffer prestabilita dal programmatore e successivamente si mette in attesa finché non arriva il messaggio di qualcuno. Una volta arrivato

creo il nuovo messaggio convertendolo in String e lo aggiungo nella lista dei suggerimenti del round

- `StopThread()`:
 - questa funzione permette di manipolare la terminazione del processo in maniera ottimale. Imposta il segnale `stop = true` quando viene chiamata dal processo `WordleClientMain`.
 - OSSERVAZIONI: Utilizzo una primitiva di sincronizzazione “synchronized”.

5.7 Giocatore:

Classe relativa al giocatore contenente le variabili utili del giocatore che serviranno successivamente per popolare il file json con le informazioni e statistiche. Ho implementato una `toString()` così da poter stampare con più comodità e visibilità le informazioni riguardanti la `Guess Distribution`.

Contiene solo due metodi:

- `GDtoString()`:
 - utile solo per stampare la `guess distribution` con i tentativi effettuati
- `IncrementAttempt(int idx)`:
 - passato l'indice come parametro, aggiorna il tentativo incrementandolo di uno nell'array dei tentativi.