

Wordle

Introduzione

Il progetto prevede la realizzazione di un gioco ispirato a Wordle, in cui gli utenti devono indovinare una parola che viene aggiornata periodicamente.

In estrema sintesi, il gioco è realizzato tramite una comunicazione client-server tra più client che inviano tentativi di indovinare la parola e un server che gestisce le richieste. Ho strutturato il mio progetto con due eseguibili main (rispettivamente ServerMain e ClientMain) la cui logica è implementata in altre classi ausiliarie. Le classi ausiliarie sono implementate come Utility classes, per le quali ho rispettato le seguenti practice: sono dichiarate come final per prevenire il sub-classing, i metodi sono static per prevenire l'override e si invocano con UtilsClass.method() , infine i costruttori sono privati per prevenire l'istanziamento di tali classi (Singleton).

La maggior parte degli oneri è a carico del server e, di conseguenza, la logica server-side è più complicata rispetto a quella del client. Per questo motivo ho due classi per svolgere le operazioni del server: GameServer, che implementa la logica del gioco lato server (a partire dalla richiesta di registrazione fino alla richiesta di logout), e UtilsServer che contiene metodi per eseguire alcune delle computazioni utili a GameServer.

ServerMain riceve le richieste dei vari client e ne delega la gestione alla classe ReqHandler, che si occupa di identificare la richiesta che gli è stata inoltrata da ServerMain e, in base al tipo di richiesta, chiamare un'opportuna funzione per svolgere il lavoro.

Vediamo tutte le scelte implementative. Seguirà, per ogni classe, un breakdown dei metodi presenti e del workflow.

Config files

Come previsto dal testo, sia il server che il client leggono le impostazioni da una file di configurazione. In questo progetto `user_settings.json` è il file per le impostazioni del client e si ha la possibilità di configurare i seguenti parametri: username, password, ip, port, multicast address, multicast port, numero di massimo di tentativi di login, timeout per socket Communication. In `server_settings.json` si possono configurare: port, multicast address, multicast port, tempo che intercorre tra una parola e l'altra (in secondi)

ClientMain

Come prima cosa il client legge le impostazioni dal file di configurazione e setta i valori letti. Dopodiché apre le risorse con un try-with-resources, in particolare le risorse di cui fa uso sono:

- Socket per la comunicazione con il server
- PrintWriter per inviare richieste al server tramite socket
- BufferedReader per leggere le risposte del server
- Scanner per leggere le parole da userInput (e, a fine gioco, le scelte di rigiocare - condividere il risultato - visualizzare la classifica - visualizzare le notifiche
- MulticastSocket per unirsi al gruppo sociale (realizzato con il multicast) e ricevere le notifiche

Ancora come fase preliminare prima di iniziare una partita, il client lancia un thread dedicato alla gestione di eventuali SIGINT. In caso di ricezione del segnale, la gestione prevede di effettuare il logout da Wordle e di chiudere le risorse (automaticamente grazie alla try-with-resources)

`register()` : per effettuare la registrazione il client si assicura in primis che la password sia idonea, cioè soddisfi i requisiti minimi. Nel testo è richiesto di controllare che la password inserita non sia vuota, nella mia implementazione ho scelto di estendere questo controllo e di imporre che la password sia lunga almeno 8 caratteri e contenga (almeno) un numero e un carattere speciale. Ovviamente questa condizione ingloba anche la richiesta del testo, cioè che la pw non sia vuota. Ho implementato questi controlli lato client, per

togliere un minimo di oneri al server, infatti finché la password non soddisfa i requisiti il client non invia la richiesta di registrazione al client. Se la registrazione fallisce, l'utente può reinserire la password da user input, nel caso avesse sbagliato ad inserirla nel file di configurazione

``login()`` : se esiste già un utente con lo stesso username, il programma procede allo step di login, verificando se la password letta dal file corrisponde con quella presente nella lista degli utenti registrati. In caso di login failed (wrong password), l'utente ha un numero massimo di tentativi prima di essere rate limited e di terminare l'esecuzione. Anche questi controlli sono effettuati lato client. Dopo login successful l'utente si unisce al gruppo sociale (multicast) e inizia la partita.

``playWORDLE()`` : durante una partita il client può effettuare al massimo 12 tentativi di indovinare la Secret Word, con l'agevolazione che se la parola inserita non è di 10 caratteri oppure non è compresa nel vocabolario allora il tentativo non viene conteggiato. Dopo ogni tentativo il client riceve come risposta dal server la parola con i caratteri colorati (come nel gioco originale). Ho preferito questa soluzione rispetto a ``verde: {}`, ``giallo: {}`, ... `` perché la avevo già usata in passato ed è molto semplice da realizzare con le stringhe ANSI, oltre che più apprezzabile esteticamente. In pratica è sufficiente definire una stringa con un determinato ANSI code (che equivale ad un colore) e concatenarla alla stringa desiderata. Quando non si vuole più usare un colore è necessario concatenare la stringa che resetta il colore (default). Ho implementato tutta questa manipolazione lato server, in GameServer.checkGuess().`

Alla fine della partita il client riceve, sia in caso di vittoria che sconfitta, la traduzione della SW e le statistiche [``sendMeStatistics()`].

Infine, ho fatto in modo che l'utente abbia la scelta di: condividere il proprio risultato relativo alla partita appena terminata tramite ``share()` , visualizzare la classifica corrente tramite ``showMeRanking()` , ricevere le notifiche delle partite degli altri utenti (per la stessa parola di cui ha appena concluso la partita) tramite ``showMeSharing()` . Per quanto riguarda le notifiche, la mia scelta è stata di visualizzare solo le notifiche relative alle partite con SW uguale a quella della partita appena terminata. Il testo è molto generico a riguardo e cita: "mostra sulla CLI le notifiche inviate dal server riguardo alle partite degli altri utenti" . Basandosi su quanto specificato dal testo, sarebbe stato più semplice (e comunque corretto) visualizzare lo storico delle notifiche senza implementare nessun filtraggio, ma ho scelto di filtrare le notifiche in

base alla partita corrente in quanto mi sembra l'opzione più verosimile. In uno scenario reale un utente non è interessato a vedere tutte le notifiche di tutti gli utenti per tutte le partite da essi effettuate, piuttosto è più verosimile che l'utente abbia appena terminato una partita e sia incuriosito di paragonare il risultato appena ottenuto con quello dei suoi amici.

Infine, il client ha la possibilità di iniziare una nuova partita: se sceglie di sì, invia al server una richiesta specifica per sapere se la SW è stata aggiornata. Se è ancora la stessa, l'utente resta in attesa e controlla periodicamente con un polling se la SW è cambiata. Un altro modo in cui avrei potuto realizzarlo senza effettuare interrogazioni periodiche al server era di fare in modo che ad ogni aggiornamento della SW il server inviasse un messaggio multicast. Ho evitato questa soluzione perché l'aggiornamento della SW non è un'informazione che interessa a tutti gli utenti (sono interessati solo i giocatori in attesa che venga aggiornata, quelli che stanno già giocando non vogliono saperlo), quindi come trade-off ho preferito pagare l'overhead di qualche richiesta in più, senza inondare il gruppo sociale con messaggi inutili. Se invece l'utente non vuole rigiocare, viene effettuato il `logout()`

UtilsClient

UtilsClient è una classe che contiene principalmente metodi per l'invio di messaggi al server.

- `parseConfig()` permette di leggere le opzioni dal file `user_settings.json` tramite un `JsonReader` e utilizzare le impostazioni desiderate dall'utente
- `getUpdates()` viene chiamato da `ClientMain` quando, al termine di una partita, il giocatore sceglie di rigiocare ma non può farlo immediatamente perché la parola da indovinare non è ancora stata aggiornata. In questo caso il giocatore non viene kickato, bensì ho implementato una specie di polling tramite questa funzione. Il client invia periodicamente (default ogni 10s) richieste al server per sapere se la parola è stata aggiornata oppure no. Se è cambiata, il giocatore può iniziare una nuova partita, altrimenti viene rimesso in sleep
- `register()` effettua un controllo sulla validità della password, che deve rispettare i requisiti minimi di 8 caratteri e (almeno) un numero e un carattere speciale. Se la password viene convalidata, il client invia al server un messaggio del tipo `[REGISTER] username password` e il server si occuperà di perfezionare la registrazione. Vedremo in seguito come funziona il protocollo per l'identificazione e la gestione delle richieste

provenienti dal client. Se invece la password non è valida, viene data all'utente la possibilità di reinserirne una valida.

- `handleSIGINT()` è il metodo per intercettare e catturare eventuali segnali di interruzione. La funzione è implementata in modi diversi per client e server, in quanto un'interruzione brusca (`sigint`) lato server è un evento molto più dannoso rispetto ad un'interruzione su un singolo giocatore. Lato client, ciò che viene fatto per la gestione del segnale è inviare al server una richiesta di logout e chiudere le risorse (fatto automaticamente grazie alla `try with resources`)
- `login()` , `logout()` , `playWORDLE()` , `sendWord()` , `sendMeStatistics()` , `share()` , `showMeSharing()` , `showMeRanking()` ` sono tutte funzioni implementate in modo analogo, che consistono nell'inviare al server una richiesta in cui il prefisso (prima parola della stringa) consente al server di individuare il tipo di richiesta. Questa implementazione fa parte del protocollo per l'identificazione e la gestione delle richieste; una descrizione accurata del protocollo è fornita in seguito, nel paragrafo `ReqHandler`

Multicast

La classe `Multicast` contiene i metodi `sendMulticast()` , utilizzato dal server, e `joinGroup()` , utilizzato dai vari client. Tramite `sendMulticast()` il server invia le notifiche agli utenti. Le notifiche inviate dal server possono essere di tre tipi:

- aggiornamento della classifica (in seguito ad un cambiamento nelle prime tre posizioni)
- condivisione di un utente del risultato della sua partita sul gruppo sociale
- Messaggio `server shutdown` , quando il server riceve un `SIGINT`. Alla ricezione del segnale, il server invia il messaggio sul gruppo sociale tramite il multicast, per comunicare a tutti gli utenti connessi che il server è stato chiuso. Di conseguenza i client chiudono le socket e terminano

Ogni volta che `sendMulticast()` invia una notifica, la salva anche nell' array `notificationsArray` e le mantiene ordinate in base al timestamp, dalla più vecchia alla più recente. Infine le scrive sul file `notifications.json`

`joinGroup()` è la funzione invocata da ogni client dopo il login, per unirsi al gruppo sociale e ricevere notifiche. Per ogni utente, `joinGroup()` lancia un thread che riceve e stampa i messaggi multicast, usando un buffer per la ricezione avente dimensione 8192byte (8kB). 8k è decisamente overkill ma mi permette di gestire anche messaggi di grandi dimensioni, ad esempio quando viene richiesto di ricevere le notifiche per un grande numero di amici che hanno condiviso il loro risultato nel gruppo sociale. Una soluzione più flessibile sarebbe stata inviare un messaggio preliminare che indica la dimensione del messaggio vero e proprio, per poi allocare la memoria sufficiente e ricevere il messaggio senza sprecare memoria extra.

ServerMain

Il flusso di lavoro del server inizia con la lettura e il salvataggio delle impostazioni dal file ``server_settings.json``. Più dettagliatamente: numero di porta, indirizzo multicast, porta multicast, thread pool size, intervallo di tempo che passa tra l'estrazione di una parola e la successiva (`next_word``), espresso in secondi. Dopo aver settato le configurazioni, il server chiama la funzione ``UtilsServer.fillWordsMap()``, che legge il file testuale `words.txt` e inserisce nella hash map tutte le parole presenti nel vocabolario, in modo da essere agevolato per gli accessi successivi. Fatto ciò, il server estrae in modo casuale la parola da indovinare tramite la funzione ``pickWord()``. La parola da indovinare viene aggiornata periodicamente da un timer thread, cioè un thread che ogni `<new_word>` secondi esegue la routine di aggiornamento della parola. Nello specifico un timer thread schedula l'esecuzione di un task con un delay prefissato (`fixed-delay execution``). La funzione ``schedule()`` ha come argomenti un task da eseguire, un delay che indica dopo quanto verrà effettuata la prima esecuzione del task, e un periodo che indica ogni quanto verranno effettuate le esecuzioni successive. Ogni esecuzione è schedulata relativamente all'esecuzione precedente, quindi se un'esecuzione viene ritardata per qualche motivo (e.g. garbage collection) allora anche tutte le successive risulteranno in ritardo.

Dopo queste operazioni preliminari, il server viene avviato insieme all'apertura di altre risorse quali la server socket, la socket multicast e la thread pool. Per quanto riguarda la thread pool, ho scelto di utilizzare una fixed-size pool per imporre un upper bound al numero di thread che possono essere creati. In questo modo evito di creare thread on demand per ogni utente che avvia una partita e di mandare in sovraccarico il server durante periodi di maggiore affluenza*. Al crescere del numero di utenti collegati, il risparmio sull'overhead è notevole: infatti ad ogni utente corrispondono 3 thread, uno per gestire la connessione, un handler per intercettare eventuali sigint e uno per la ricezione di messaggi sul gruppo sociale.

* ovviamente stiamo ragionando nell'ottica di questo progetto e di quanto abbiamo visto nel corso, infatti in una situazione reale sarebbe impensabile hostare un server in locale, bensì si ricorrerebbe a soluzioni scalabili ad esempio virtual machines molto più potenti del semplice laptop su cui hostiamo i nostri programmi client-server sviluppati durante il corso

Dopo queste operazioni il server si mette in ascolto di nuove connessioni finché non riceve un eventuale segnale sigint. Il server infatti non termina mai spontaneamente, termina solo in seguito alla ricezione di un segnale di interruzione. Per ogni connessione con un client, il server la inoltra ad un thread handler che si occupa di gestirla. La gestione delle richieste è dunque delegata. Vediamo come funziona nel prossimo paragrafo

ReqHandler

La classe ReqHandler ha il compito di gestire le connessioni del server. Ogni connessione è gestita da un thread che riceve le richieste del client sulla socket e identifica la richiesta in base al messaggio ricevuto. Ho scelto di implementare un protocollo molto semplice per la comunicazione, che funziona nel modo seguente: ogni messaggio inviato dal client tramite socket è anticipato da un prefisso che ne specifica il tipo, e.g. `[LOGOUT] dummyUser`. Il thread gestore delle richieste, per ogni messaggio, fa una split della stringa e in base al token in posizione 0 riesce ad identificare di che tipo di richiesta si tratta. Poi, tramite uno switch, invoca una funzione specifica in base al tipo di richiesta. In questo esempio il messaggio ricevuto è `[LOGOUT] dummyUser` quindi la funzione `handleReq` analizza la prima parola del messaggio e fa uno `switch(exp)` con `exp = request.split(" ")[0]`, cioè la prima parola della richiesta. In questo modo ha identificato il tipo di richiesta e invoca l'apposita funzione `logout(username)`. Questa procedura vale per qualsiasi richiesta inviata dal client.

GameServer

In questa classe sono implementati tutti i metodi che compongono ogni step del gioco, a partire dalla fase di login fino al logout dell'utente. Vediamoli adesso uno per uno.

- **`register`** è la funzione che esegue la registrazione di un determinato utente al servizio Wordle. Come prima cosa controlla, tramite ``isRegistered``, che l'utente non sia già registrato e in caso lo fosse si passa direttamente al login. Se invece non è già registrato viene effettuata la registrazione chiamando ``storeInfo`` che salva le credenziali dell'utente nel file ``users.json``. Dopo la registrazione, il server invia al client una risposta per confermare che la registrazione è stata eseguita con successo.
- **`login`** controlla, tramite la funzione ``checkLogin``, che tra gli utenti salvati nel file `users.json` ci sia un utente avente `user:pass` che matchano quelli ricevuti dal client. In tal caso, il server risponde alla richiesta di login con un messaggio di login successful, altrimenti login failed e il client avrà `<max_attempts>` tentativi (valore letto da `user_settings.json`) per ritentare il login.
- **`playWordle`** è la funzione per iniziare una nuova partita. Innanzitutto il server effettua un controllo per impedire all'utente di avviare due partite per la stessa parola. Questo controllo è realizzato con l'ausilio di una `ConcurrentHashMap `games`` avente come key:value entries `username (String)` e `playedWords (ArrayList<String>)` *. Grazie alla hash map si accede in tempo costante all'array delle parole giocate da ogni utente e in $O(n)$ si stabilisce se una parola è già stata giocata dall'utente oppure no.

Se l'utente ha già disputato una partita con tale parola, il server invia un messaggio tramite socket in cui dice di riprovare più tardi. In caso contrario, il server memorizza nella hash map `games` il nome dell'utente e l'array delle parole che ha giocato e scrive i dati aggiornati nel file `users.json`

- * ho usato una `ConcurrentHashMap` per gestire le scritture concorrenti nella map.
- **`sendStats`** è il metodo che viene chiamato al termine di ogni partita, sia terminata vittoriosamente sia con sconfitta. Il server invia al client un messaggio contenente le sue statistiche, visualizzato come json object (prettified)
- **`sendRanks`** è il metodo invocato quando il client decide (opzionalmente) di visualizzare la classifica corrente, i.e. quando il client invoca il metodo ``showMeRanking``. Per fare ciò, il server prende ogni object (che corrisponde ad un utente) dal `JSONArray `leaderboardArray``, converte il `JsonObject` in stringa e lo concatena ad una stringa che sarà il messaggio da inviare al client, cioè la classifica. Ovviamente il sorting non viene effettuato in questo step perché all'interno di `leaderboardArray` gli utenti sono già ordinati in base allo score (dal più basso al più alto perché per come calcoliamo noi lo score vale che "lower is better", come suggerito da Matteo Loporchio), dunque questo metodo lavora su valori già ordinati. Per leggibilità, non ho costruito la classifica prendendo il `JsonObject` intero per ogni utente ma ho preferito mostrare solo username e score nel seguente formato (senza il trattino dash -
 - 1. Gianni (3.456)
 - 2. Gianna (4.567)
 - 3. Mario (5.678)In `leaderboardArray` e in `leaderboard.json` sono presenti informazioni più complete riguardo la classifica, in particolare per ogni utente mostro nella classifica i seguenti dati: username, score, numero di vittorie, ratio di vittorie:sconfitte espresso in numero percentuale (win rate), numero di sconfitte, numero di partite giocate, numero medio di tentativi, streak corrente, streak massima raggiunta, guess distribution (come indicato da Matteo Loporchio)
- **`sendShares`** è il metodo invocato quando il client decide (opzionalmente) di condividere sul multicast il risultato della partita che ha appena terminato, i.e. quando il client invoca il metodo ``showMeSharing``. In questo caso il server seleziona dall'`ArrayList `notifications`` tutte le condivisioni delle partite con quella parola, disputate dagli altri utenti (quindi le condizioni su cui filtro sono sia "username diverso dall'utente che ha richiesto lo `showMeSharing``", sia che "la parola è uguale a quella della

partita per cui è stato richiesto lo showMeSharing”). Il messaggio da inviare al client è una stringa costituita dalla concatenazione delle notifiche nel seguente formato:

<parola> | from: <username>



Per esempio, questo è il caso in cui un utente indovina al secondo tentativo. Se nessun utente ha ancora condiviso il suo risultato per quella determinata parola, il server invia come risposta il messaggio “Nobody has yet shared their results for the word <parola>”

- ``logout``: quando un utente chiede di sloggarsi, il server rimuove l'utente dalla `ConcurrentHashMap `sessions`` delle sessioni attive. La terminazione della partita viene effettuata client-side e il client chiude la connessione.
- ``updateLeaderboard`` è un metodo `synchronized` in quanto prevede una scrittura sul file `leaderboard.json` e, in presenza di più giocatori, è probabile che si verifichino scritture concorrenti sul file. Il metodo esegue l'aggiornamento della classifica al termine di ogni partita, aggiornando i dati menzionati poco sopra nella descrizione di ``sendRanks``. ``updateLeaderboard`` effettua anche un controllo sulle prime tre posizioni della classifica: se sono cambiate in seguito all'aggiornamento (e se la classifica è composto da minimo 3 giocatori), il server invia un messaggio sul multicast a tutti i giocatori connessi, con cui segnala il cambiamento nelle prime tre posizioni specificando i top3 player attuali. Il formato di questo messaggio è il seguente: “Leaderboard changed: 🏆 player1 | 🥈 player2 | 🥉 player3”




UtilsServer

UtilsServer è una classe contenente metodi che calcolano i risultati necessari alle funzioni di GameServer. La classe UtilsServer è dichiarata come `final` per prevenire il sub-classing, i metodi sono `static` per prevenire l'override e si invocano con `UtilsServer.method()`, infine il costruttore è privato in modo da prevenire l'istanziatura della classe. Vediamo adesso i metodi uno per uno:

- ``sortJsonArray()`` ordina un `JSONArray` in base alla property “`sortBy`” che gli passiamo come argomento. Nel progetto ho usato questo metodo in due casi: per ordinare le notifiche di `notifications.json` in base al timestamp (dalla meno recente alla più recente) e per ordinare la classifica in `leaderboard.json` (dallo score più basso al più alto, come spiegato in precedenza).

- `computeScore()` si occupa, prevedibilmente, di calcolare il punteggio aggiornato di un giocatore al termine di una partita. Per il calcolo del punteggio ho seguito la guida fornita da Matteo Loporchio, tranne per un dettaglio ambiguo. La spiegazione fornita cita: “Per calcolare il punteggio, moltiplico la posizione *i*-esima del vettore per *i* stesso. Sommo tutti i prodotti e divido per il numero totale di partite giocate. Nella somma, tengo conto anche delle parole non indovinate (partite perse). In questo caso, considero un numero di tentativi pari al numero massimo + 1”
Nell’esempio che è stato fornito insieme alla spiegazione non si tiene conto del fatto che in posizione 0 dell’array della `guessDistribution` ci sarà necessariamente sempre e solo il valore 0, in quanto per vincere è necessario fare come minimo 1 tentativo. Infatti in caso il giocatore riuscisse ad indovinare al primo tentativo, verrebbe incrementato di 1 il valore `guessDistribution[1]`, mentre il valore `guessDistribution[0]` rimane sempre fisso a 0. Questa è l’unica differenza della mia implementazione, per il resto ho seguito lo stesso principio per il calcolo del punteggio
- `setConfig()` legge le impostazioni dal file `server_settings.json` tramite `JsonReader` e ne assegna i valori alle variabili di configurazione, nello specifico: numero di thread della pool, intervallo di tempo tra una parola e l’estrazione della successiva (espresso in secondi), numero di porta, indirizzo multicast, porta multicast.
- `fillWordsMap()` legge il file testuale `words.txt` e inserisce le parole del vocabolario nella hashmap `wordsMap` per facilitare gli accessi successivi, ad esempio per ogni tentativo di ogni giocatore il server deve controllare se la parola ricevuta è presente nel vocabolario. Quindi è un’operazione molto frequente che, grazie ad una hashmap, può essere eseguita in tempo costante
- `fillUsersMap()` legge il file `users.json` e inserisce gli utenti (come `JsonObjects` aventi lo username come chiave) nella hash map `usersMap`. Questo avviene dopo ogni (ri)avvio del server, in quanto tutte le operazioni relative agli utenti vengono svolte per convenienza sui dati estratti dalla hash map grazie a ricerche e inserimenti in tempo costante. Altre considerazioni sono presenti più in dettaglio nella sezione dedicata alle strutture dati.
- `pickWord()` sceglie una parola casuale dal vocabolario. Non lo fa leggendo direttamente il file, bensì tramite la hashmap. In particolare genera un numero casuale, poi converte l’insieme delle chiavi della mappa (`map.keySet()`) in un array e infine estrae dall’array la stringa all’indice corrispondente al valore casuale generato. Questa operazione, come già detto in precedenza, viene svolta ogni `<next_word>` secondi, con `<next_word>` valore letto dal file di configurazione `server_settings.json`
- `getTranslation()` viene chiamata al termine di ogni partita (conclusasi sia con vittoria che con sconfitta) per inviare al giocatore la traduzione della parola da indovinare. Questo metodo consiste nell’invio di una GET request ad un url costruito partendo da un indirizzo base a cui ho concatenato i parametri per la query (la parola di cui ci interessa la

traduzione). Infine, dal json della response estraggo il campo `responseData` che corrisponde alla traduzione della parola.

- `checkGuess()` serve per analizzare la Guess word inviata dall'utente e costruire il messaggio da inviare come risposta. Come prima cosa verifica se il tentativo esiste tra le parole del vocabolario. In caso affermativo, procede controllando quali lettere corrispondono, seguendo le regole classiche di Wordle che già conosciamo. Altrimenti invia al client una risposta per segnalare che la parola inviata non esiste, e di conseguenza il client effettuerà un nuovo tentativo senza conteggiare il tentativo per la parola che non esisteva (come previsto dal testo del progetto). Se il server dovesse, per ogni tentativo effettuato da ogni giocatore, leggere il file testuale `words.txt` e cercare la parola inviata dall'utente (a volte anche senza successo, quindi con complessità esattamente $\Theta(n)$ per lo scan di tutte le parole) sarebbe un'operazione molto costosa. Per effettuare ricerche in tempo costante ho deciso di usare una hash map `wordsMap` che permette al server di stabilire se il tentativo di ogni giocatore esiste nel vocabolario con complessità $O(1)$. Non ho dichiarato la map come concurrent perché viene usata solo per letture quindi non c'è bisogno di gestire la concorrenza. In seguito viene incrementato di 1 il numero di tentativi del giocatore relativamente alla partita corrente e poi si passa alla costruzione della stringa da inviare come risposta al tentativo del giocatore. Se il guess corrisponde alla parola da indovinare, il server aggiorna la classifica, richiede la traduzione della parola e le statistiche aggiornate del giocatore, invia al client un messaggio costituito dalla parola indovinata, dalla sua traduzione e il risultato finale "no spoiler" costituito dalle emoji nere/gialle/verdi; dopodiché invia anche le statistiche. Se invece il guess non è esatto, si aggiorna le stringhe (man mano dopo ogni tentativo) per la condivisione social e del messaggio di risposta da inviare al client. Per la costruzione di queste due stringhe si segue il regolamento illustrato nel testo del progetto: per ogni carattere 'c' della stringa guess, se c non è presente nella parola da indovinare aggiungo c , senza alcun colore, alla stringa "result" e aggiungo  alla stringa "social" ; se c è presente ma non nella posizione corretta aggiungo c , colorato di giallo, alla stringa result e aggiungo  alla stringa social ; se c è presente e si trova nella stessa posizione in cui è presente nella parola da indovinare, allora aggiungo c verde alla stringa result e  alla stringa social. Il colore alle stringhe è realizzato tramite ANSI code (vd. spiegazione del metodo `ClientMain.playWORDLE()`).

Strutture dati

- ``usersArray`` : è un `JSONArray` che uso per leggere e scrivere dati dal file `users.json`. La lettura avviene al momento in cui si avvia il server: i dati del file `users.json` vengono ricopiati in `usersArray`, poi durante le partite si effettuano modifiche all'array (e.g. in seguito alla registrazione di nuovi utenti) e si riscrive il `JSONArray` aggiornato sul file `users.json`. Per le operazioni di ricerca - ad esempio quando devo recuperare una proprietà di un utente - uso una hash map che mantengo sempre aggiornata. I `JSONArray` non offrono metodi built-in per la gestione della concorrenza quindi ogni volta che devo accedere a `usersArray` uso un blocco `synchronized` per garantire che la struttura dati sia usata da un solo thread alla volta.
- ``usersMap`` è la `ConcurrentHashMap` che uso in ausilio a `usersArray`. Come chiave utilizzo lo username dell'utente e come value un `JsonObject` contenente tutte le informazioni rilevanti. Per quanto riguarda i dati sulle partite disputate, in questa map tengo traccia solo delle parole per cui l'utente ha già giocato, mentre le statistiche sulle partite effettuate le memorizzo in un'altra struttura dati che vedremo tra poco. La `usersMap` risulta utile, ad esempio, quando devo stabilire se un utente è già registrato oppure no, infatti posso effettuare l'operazione in tempo costante anziché fare una scansione completa di tutto il `JSONArray`. Il drawback è che ad ogni spegnimento del server perdo i dati della map, quindi al momento dell'avvio del server è necessario leggere il file `users.json` (che memorizza le informazioni in modo permanente) ed inserire le informazioni sulla map. Quello che conta in questo caso è la frequenza: infatti l'overhead per la riscrittura completa sulla map è un evento raro (solo in caso di riavvio del server), mentre è molto più frequente la ricerca di un utente tra la lista degli utenti registrati. In sintesi, è un trade-off nettamente a favore della hash map nonostante le riscritture necessarie in caso di riavvio del server.

Il file in cui memorizzo le informazioni sugli utenti registrati è ``users.json``. Il contenuto è un `JSONArray` avente come elementi dei `JsonObject`, ciascuno costituito da: `username`, `password`, `playedWords`.

Come già accennato, i dati in `usersArray` e `usersMap` vengono persi in caso di spegnimento del server quindi, ad ogni avvio, il server legge e ricopia gli utenti da `users.json` per poi accedervi (durante tutto il lifecycle del server) in tempo costante tramite hash map.

- ``leaderboardArray`` è un `JSONArray` che uso letture e scritture sul file `leaderboard.json`. Analogamente a `usersArray`, `leaderboardArray` contiene dei `JsonObject`. Ognuno di essi corrisponde ad una posizione della classifica, nonché un oggetto json contenente username e dati rilevanti per la classifica, in particolare: `username`, `sconfitte`, `vittorie`, `numero di partite giocate`, `numero medio di tentativi`, `streak corrente di vittorie`, `massima streak di vittorie`, `guess distribution`, `rateo vittorie:sconfitte (win rate)` e

- infine il punteggio dell'utente. Per gli accessi e le modifiche concorrenti a al JSONArray leaderboardArray vale quanto detto per usersArray: ogni accesso all'array è effettuato all'interno di un blocco synchronized.
- `leaderboard.json` non è altro che il risultato delle scritture di leaderboardArray. E' un file che consiste in un JSONArray con tutti gli oggetti e le proprietà elencate sopra per leaderboardArray. Come per users, il file json permette di memorizzare i dati in modo persistente mentre il json array e le hash map vengono usati durante la gestione delle partite per lavorare con i dati. Anche in questo caso, dopo un (ri)avvio del server si ricopiano i dati dal file json alla struttura dati. Ho già analizzato prima (per users) la convenienza di questo approccio. Per quanto riguarda il salvataggio e l'aggiornamento della classifica, il server la mantiene ordinata dal punteggio più basso al più alto (come spiegato da Matteo Loporchio) e ad ogni aggiornamento controlla se c'è stato un cambiamento nelle prime tre posizioni e in tal caso invia un messaggio sul gruppo sociale con cui segnala l'evento.
 - `games` è una ConcurrentHashMap avente come chiavi gli username dei giocatori e come valori l'array, per ogni giocatore, delle parole per cui ha disputato una partita. In usersMap, per ogni value (JsonObject) ho un campo playedWords che è un array delle partite salvato come stringa. `games` dunque è utile per controllare se l'utente che richiede di giocare ha già effettuato una partita oppure no: questo accade all'inizio della partita (playWORDLE) ed eventualmente se al termine della partita sceglie di giocare di nuovo. In questo caso, come già spiegato, l'utente tramite un polling aspetta che la parola da indovinare venga aggiornata ma tra una sleep e l'altra il client invia una richiesta e il server controlla se la parola corrente è presente nella hash map games con chiave l'username del giocatore.
 - `wordsMap` è una HashMap (non Concurrent, perché non ci sono scritture concorrenti) che uso, quando (ri)avvio il server, per memorizzare tutte le parole del vocabolario (file testuale words.txt). Questa struttura è fondamentale per il progetto in quanto la ricerca delle parole nel vocabolario è un'operazione molto frequente e relativamente costosa. Si può fare in tempo logaritmico con una ricerca binaria dato che le parole sono in ordine alfabetico, ma con una hashmap si fa in tempo costante. Con questa soluzione, ogni volta che un giocatore invia un tentativo di indovinare la parola e il server deve controllare se la parola ricevuta esiste nel vocabolario, può farlo in $O(1)$
 - `socialScore` è una ConcurrentHashMap avente come chiavi gli username degli utenti e, per ognuno, memorizza come value una stringa contenente il risultato "no spoiler" che potrà condividere a fine partita. Due cose sono da considerare: la stringa è costruita iterativamente e aggiornata dopo ogni tentativo dell'utente, ma viene scritta nella map solo a fine partita. La costruzione della stringa, come abbiamo visto, è ispirata al gioco originale utilizzando le emoji anziché verde={...}, giallo={...}, grigio={...} La seconda cosa di cui si deve tener conto è che si riferisce al risultato "no spoiler"

- della partita corrente, quindi se al termine di una partita l'utente rifiuta di condividere il suo risultato non potrà più farlo in un secondo momento.
- ``sessions`` è una `ConcurrentHashMap` per gestire le sessioni attualmente attive e il numero di tentativi effettuati da ogni giocatore. Ha come chiave l'username del giocatore e come value un intero che indica il numero di tentativi effettuati nella partita corrente. La sessione viene chiusa (rimossa dalla map) quando l'utente effettua il logout
 - ``notifications`` è un array di stringhe usato per raggruppare le notifiche da inviare quando un utente richiede, a fine partita, di ricevere le notifiche degli altri giocatori che hanno condiviso il proprio risultato per la parola corrente. Tutte le notifiche sono comunque salvate su un file json e ordinate in base al timestamp, dalla più vecchia alla più recente. Esistono sono scenari in cui più thread modificano l'array, quindi ho definito l'array `notifications` come thread safe: ``static Collection<String> notifications = Collections.synchronizedList(new ArrayList<String>());``
 - ``notifications.json`` è il file in cui sono salvate in modo permanente tutte le notifiche: condivisioni di partite, messaggi di server shutdown (nel paragrafo Multicast abbiamo visto come funziona), aggiornamenti dei primi tre posti della classifica. Ogni notifica corrisponde ad un `JsonObject` e ogni tipo di notifica ha Properties differenti: per i messaggi di server shutdown interessano solo timestamp e contenuto del messaggio, per le condivisioni social interessano username, timestamp, parola e il "risultato no spoiler" ; infine per un cambiamento nelle prime tre posizioni viene memorizzato il timestamp e il contenuto del messaggio, che specifica la nuova top 3 della classifica

Istruzioni per l'utilizzo

Per la compilazione dei file java è sufficiente aprire la cartella Wordle sul terminale, e da qui eseguire il comando `javac -cp ./lib/gson-2.9.1.jar -d ./bin/ @./settings/sources.txt` per compilarli tutti in un colpo solo.

In breve, specifico il classpath delle librerie esterne, nel mio caso solo la libreria gson, e il path dei file .java viene letto, per ogni file, da sources.txt. Il comando crea inoltre una cartella "bin" contenente i .class files. sources.txt è un file testuale che ho generato tramite il comando `find ./src/ -type f -name "*.java" > ./settings/sources.txt`, che esegue la scrittura del path di ogni .java sul file testuale indicato come target.

Per eseguire i file jar si usa il comando `java -jar --enable-preview`

ServerWordle.jar per il server e `java -jar --enable-preview`

ClientWordle.jar per il client . Le funzionalità di preview sono state introdotte a partire da Java14, quindi utilizzando una versione di Java precedente alla 14 dovrebbe essere possibile eseguire il file jar senza dover utilizzare il flag `--enable-preview`, indipendentemente dalla versione di Gson. Le versioni di Gson precedenti alla 2.8.0 non utilizzano funzionalità di preview. Ad esempio Gson 2.7.0 e precedenti non utilizzano funzionalità di preview e dovrebbero essere compatibili con le versioni precedenti di Java senza il flag `--enable-preview`. Tuttavia utilizzare una versione precedente di Gson limita alcune funzionalità, ad esempio con la 2.7.0 non ero in grado di usare la `deepCopy()` su un `JsonObject` e quindi ho preferito usare la 2.9.2 senza sacrificare le funzionalità della api e aggiungendo il flag della preview da linea di comando.