Progetto Laboratorio 3 - **WORDLE: un gioco di parole 3.0**

Connessione al server

Il Server dopo aver registrato un **ServerSocketChannel** con il selettore per l'operazione di accettazione (**SelectionKey.OP_ACCEPT**), si mette in ascolto sul selettore.
Quando una nuova richiesta di accettazione viene elaborata (un client chiede di connettersi) la SocketChannel della nuova connessione viene anche essa registrata con il selettore per l'operazione di lettura (**SelectionKey.OP_READ**).
Tutte le richieste che arrivano al server vengono gestite multiplexando i thread disponibili sui canali che vengono indicati dal selettore.

Una volta creata la connessione viene memorizzata all'interno di una **ConcurrentHashMap<String, Client>**, dove la stringa è la **SocketAddress** covertita in stringa (quindi il formato sarà :), questo garantisce l'univocità della stringa all'interno di della Map.

Questo implica che per ogni client che si connette al server bisogno memorizzare delle variabili che indichino lo stato della connessione (se è registrato o longato) e della partita (numero di tentativi effettuati, se ha indovinato la Secret Word e l'ultima Secret Word per cui ha partecipato al gioco).

Per mantenere la consistenza dei dati (sopratutto quando il server viene riavviato) tutte le informazioni del utente vengono salvate all'interno del file nella directory **./resources/users/< username >.json**. Il mantenere un file json per ogni utente permette di verificare facilmente se l'utente è registrato, se le credenziali corrispondono a quelle che ha inserito in fase di registrazione e sopratutto di ricaricare in memoria lo stato delle vecchie variabili di gioco.
Questa è la struttura del file json che rappresenta un utente:

```
{
  "credentials": "ZnJhOmZyYQ\u003d\u003d",
  "lastWord": "polyadenia",
  "hasFinishMatch": false,
  "statistics": {
    "playedMatches": 43,
    "winPercentage": 0.0,
    "wonMatches": 18,
    "streakWin": 0,
    "maxStreakWin": 4,
    "guessDistribution": [7, 8, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0],
    "currentAttemps": 12
  }
}
```

> Da notare che le credenziali vengono scritte utilizzando la codifica **Base64**.

Quando un client tenta di effetuare il **login** viene effetuato un controllo per verificare che non ci sia gia un'altro client collegato con lo stesso **username**.
Questo insieme al fatto che una volta che il client invia una richiesta si mette in attesa della risposta, permette di assumere che non ci saranno mai più thread che tentano di modificare i dati di un utente contemporaneamente.

Formato delle richieste

Tutta la logica gestine delle richieste e restituzione delle risposte viene gestita dalla classe **Handler** che a sua volta si occupa di chiamare gli opportuni metodi della classe Client qualora le richeiste (rapresentate tramite la classe **Request**) sia ben formate. La classe Handler si occupa anche della restituzione delle risposte (rapresentate tramite la classe **Response**).

Questo il formato per delle richieste e delle eventuali risposte:

> ***NOTA : questa sezione soprattuto per quanto rigurada le rispsote non è esaustiva ma riporta solamente i messaggi più complessi che il server potrebbe restituire***

> **register( )**:
>
> ```
> {
>   "method": "register",
>   "credentials": "YWRtaW46YWRtaW4\u003d" // stringa codificata in
> Base64 con il formato <username>:<password>
> }
> ```
>
> risposta:
>
> ```
> {
>   "status": 200, // codice di stato della risposte (come per le
> richieste HTTP)
>   "statusMessage": "OK", // messaggio relativo allo stato
>   "message": "Ora sei registrato" // contiene un messaggio utile da
> restituire come feedback al giocare
> }
> ```

login( ):

```
{
  "method": "login",
  "credentials": "b2N0YXZpYW46b2N0YXZpYW4\u003d" //stringa codificata
in Base64 con il formato <username>:<password>
}
```

risposta:

```
{
  "status": 405, // codice di stato della risposte (come per le
richieste HTTP)
  "statusMessage": "Method Not Allowed", // messaggio relativo allo
stato
  "message": "Sei già loggato" // contiene un messaggio utile da
restituire come feedback al giocare
}
```

logout( ):

```
{
"method": "logout",
  "credentials": "ZnJhOmZyYQ\u003d\u003d" //stringa codificata in
Base64 con il formato <username>
},
```

risposta:

```
{
  "status": 200, // codice di stato della risposte (come per le
richieste HTTP)
  "statusMessage": "OK", // messaggio relativo allo stato
  "message": "Logout effettuato con successo" // contiene un
messaggio utile da restituire come feedback al giocare
}
```

playWORDLE( ):

```
{
  "method": "playWORDLE"
}
```

risposta:

```
{
  "status": 200, // codice di stato della risposte (come per le
richieste HTTP)
  "statusMessage": "OK", // messaggio relativo allo stato
  "message": {
    "endAt": "15:17:21", //rapresenta l'orario in cui l'attuale
Secret Word scadrà, quindi ne verra scelta una nuova.
    "remainingAttempts": 10, // numero di tentativi rimanenti
    "hasFinishedMatch": false // indica se l'utente ha finito i match
attuale
  }
}
```

share( ):

```
{
  "method": "share"
}
```

risposta:

```
{
  "status": 401, // codice di stato della risposte (come per le
richieste HTTP)
  "statusMessage": "Unauthorized", // messaggio relativo allo stato
  "message": "Non sei già loggato" // contiene un messaggio utile da
restituire come feedback al giocare
}
```

sendWord( ):

```
{
"method": "sendWord",
"guessedWord": "petaloso" // parola di 10 caratteri che
},
```

risposta:

```
{
  "status": 200, // codice di stato della risposte (come per le
richieste HTTP)
  "statusMessage": "OK", // messaggio relativo allo stato
  "message": {
    // questa è una delle risposte più complesse
    // perche oltre a restituire un pattern che indica
    // il matching tra la GW e la SW
    // deve gestire anche tutti gli altri casi come:
    // - parola non presente nel dizionario ("INVALID_WORD")
    // - la secret word e cambiata mentre il client era connesso
("WORD_CHANGED")
    // - l'utente ha esaurito i tentativi a sua disposizione
("MAX_ATTEMPTS_REACHED")
    // - l'utente ha gia vinto il match attuale ("MATCH_FINISHED")

    "pattern": "G__GGY_Y__", // indica il matching tra la GW e la SW
    // Le lettere del campo pattern hanno il seguente significato:
    // G lettera che è stata indovinata e si trova nella posizione
corretta rispetto a SW
    // Y è stata indovinata, ma si trova in una posizione diversa in
SW
    // _ non compare in SW
    "endAt": "15:17:21" //rapresenta l'orario in cui l'attuale Secret
Word scadrà, quindi ne verra scelta una nuova, è presente solamente
nel caso in cui il valore del campo "pattern" è "WORD_CHANGED"
  }
}
```

sendMeStatistics( ):

```
{
"method": "sendMeStatistics"
},
```

risposta:

```
{
  "status": 200, // codice di stato della risposte (come per le
richieste HTTP)
  "statusMessage": "OK", // messaggio relativo allo stato
  "message": {
    "statistics": {
      "currentAttemps": 0,
      "playedMatches": 24,
      "wonMatches": 23,
      "streakWin": 2,
      "maxStreakWin": 18,
      "winPercentage": 0,
      "guessDistribution": [0, 14, 3, 4, 1, 0, 0, 0, 0, 0, 1, 0]
    }
  }
}
```

Sia per il **Client** che per il **Server** la serializzazione e la deserializzazione da **Json** ad istanze di oggetti e stata implementata dalla classe **Parser** che fa da wrapper alla libreria **Gson**.

Thread attivi

Nel **Client** è presente solamente un thread attivo che si occupa di leggere gli input da tastiera e di inviare le richieste al server.
Dopo la fase di **login**, viene avviato un nuovo thread che si mette in ascolto su una **MulticastSocket** per ricevere inviate dal server relative alle statistiche di gioco degli altri utenti.

Nel **Server** sono presenti almeno 3 thread attivi contemporaneamente:

- **Main thread**: avviare tutti gli altri thread e poi si mette in ascolto sul selettore, all'arrivo di nuove richieste lancia un nuovo thread (**threadPool.execute()**) per gestire la richiesta
- **Secret Word thread**: si occupa di generare una nuova Secret Word, e di segnalare alla classe Client che la Secret Word è cambiata
- **thread di terminazione**: si occupa di gestire la terminazione del server

Oltre a questi thread, ci sono anche i **thread di gestione delle richieste** che si occupano di gestire le richieste dei client e fornire le relative risposte. Il numero di questi thread è gestito utilizzando un **thread pool** di dimensione fissa, basato sulla variabile **THREAD_POOL_SIZE**.

Interruzione e chiusura server

Quando il server riceve un segnale di terminazione, il thread che si occupa della terminazione (classe **TerminationHandler**) viene eseguito e chiama:

- il metodo ***stop()*** della classe **Server** che si occupa di chiudere il selettore, chiudere la ServerSocketChannel e, infine, chiudere il thread pool.
- il metodo ***stop()*** della classe **SecretWord** che si occupa di interrompere il thread che si occupa di generare la Secret Word.
- il metodo ***logoutAll()*** della classe **Client** che si occupa di scrivere su file le statistiche di tutti i client connessi.

Primitive di sincronizzazione

Come già anticipato in precedenza, il **Server** impedisce a più client di giocare contemporaneamente con lo stesso account. Questa scelta è stata fatta per evitare race condition e inconsistenze nel gioco.
Per implementare questa funzionalità, è stata utilizzata una **ConcurrentHashMap** che mappa gli username degli utenti connessi che hanno effettuato l'accesso e avviato una partita.
Per gestire la **Secret Word**, è stata utilizzata una variabile **volatile**.

Il **Client** per gestire la ricezione delle statistiche degli altri utenti è stata utilizzata una **ConcurrentLinkedQueue** che mantiene al suo interno le statistiche degli utenti che sono state ricevute dal Client.

Compilazione ed esecuzione

Per compilare ed avviare il progetto è necessario spostarsi con il terminale nella cartella wordle/Server oppure wordle/Cliente in base a quale dei due si vuole compilare ed eseguire.
Una volta spostati nella cartella desiderata è possibile compilare ed eseguire il progetto con i seguenti comandi:

> **Client**
>
> **compilazione:**
>
> ```
> javac -cp ".:./lib/*" -d ./bin ./src/*.java
> ```
>
> **esecuzione:**
>
> ```
> java -cp ".:./bin:./lib/*" Main
> ```
>
> **compilazione ed esecuzione:**
>
> ```
> javac -cp lib/* -d bin/ src/* &&
> java -cp ".:./bin:./lib/*" Main
> ```
>
> **compilazione e creazione del file jar:**
>
> ```
> javac -cp ".:./lib/*" -d ./bin ./src/*.java &&
> jar cvfm Client.jar META-INF/MANIFEST.MF -C bin/ . -C lib/ .
> ```
>
> **Esecuzione del file jar:**
>
> ```
> java -jar Client.jar
> ```

Server

compilazione:

```
javac -cp ".:./lib/*" -d ./bin ./src/*.java
```

esecuzione:

```
java -cp ".:./bin:./lib/*" Main
```

compilazione ed esecuzione:

```
javac -cp ".:./lib/*" -d ./bin ./src/*.java &&
java -cp ".:./bin:./lib/*" Main
```

compilazione e creazione del file jar:

```
javac -cp ".:./lib/*" -d ./bin ./src/*.java &&
jar -cfvm Server.jar META-INF/MANIFEST.MF -C bin/ .  -C lib/ .
```

esecuzione del file jar:

```
java -jar Server.jar
```

Paramentri di configurazione

La lettura dei paramtri di configurazione avviene tramite la classe **Config** che si occupa di leggere il file **.properties** presente all'interno della cartella **resources** e di caricare all'interno di variabili statiche i parametri di configurazione.

Parametri del Server:

```
PLAYER_MAX_ATTEMPTS = 12
WORDS_LENGTH = 10

DEFAULT_PORT = 8000
DEFAULT_HOST = localhost
WORDS_DURATION = 1

MULTICAST_PORT = 12000
MULTICAST_GROUP = 226.1.2.3

THREAD_POOL_SIZE = 1
```

Parametri del Client:

```
PLAYER_MAX_ATTEMPTS = 12
WORDS_LENGTH = 10
WORDS_DURATION = 1

SERVER_PORT = 8000
SERVER_HOST = 127.0.0.1

MULTICAST_PORT = 12000
MULTICAST_GROUP = 226.1.2.3
MULTICAST_INTERFACE = en0
```