WORDLE: un gioco di parole 3.0

Ginevra Maoro 581700

Contenuto della cartella contenente il progetto

Nella cartella consegnata sono presenti le seguenti sottocartelle:

- **server**: contiene i codici sorgente del server: *Server.java* e *ServerMain.java* che contiene il main del server.
- **condivisi**: contiene i codici sorgenti delle classi condivise sia dal client che dal server.
- **client**: contiene i codici sorgente del client: *Client.java* e *ClientMain.java* che contiene il main del client.
- **bin:** contiene i file eseguibili, ottenuti tramite la compilazione con comando *javac* dei codici sorgenti del client e del server.

Nella cartella **GinevraMaoroWordle** sono inoltre presenti i seguenti file

- *words.txt*: file che contiene le parole del dizionario di Wordle.
- *configurazione_Server*.properties: contiene i paramenti di input dell'applicazione server come i numeri di porta, indirizzi e valori di timeout.
- *configurazione_Client*.properties: contiene i paramenti di input dell'applicazione client.
- *Registro_registrazioni*.json: file json che contiene le informazioni relative agli utenti registrati. È utilizzato per far si che le informazioni degli utenti persistano lato server.
- *ServerMain*.jar: file JAR eseguibile per l'applicazione server, è stato creato posizionandosi nella cartella bin da terminale, dopo che il sorgente *ServerMain.java* è stato compilato, eseguendo il seguente comando:
  ```
  jar cmfv ../SMANIFEST.txt ../ServerMain.jar *.class
  ```
- *ClientMain*.jar: file JAR eseguibile per l'applicazione client, è stato creato posizionandosi nella cartella bin da terminale, dopo che il sorgente *ClientMain.java* è stato compilato, eseguendo il seguente comando:
  ```
  jar cmfv ../CMANIFEST.txt ../ClientMain.jar *.class
  ```
- *SMANIFEST.txt*: file manifest usato per creare il file JAR relativo all'applicazione server.
- *CMANIFEST.txt*: file manifest usato per creare il file JAR relativo all'applicazione client
- *gson*-2.10.jar: libreria esterna utilizzata nel progetto per serializzare/deserializzare oggetti java in/da JSON

Istruzioni per compilare ed eseguire il progetto

Su macOS e Ubuntu:

Aprire il terminale e posizionarsi all'interno della cartella GinevraMaoroWordle, a questo punto **compilare** Server e Client rispettivamente con i comandi

```
javac -cp .:gson-2.10.jar -d ./bin/ -sourcepath ./server/:./condivisi/
./server/ServerMain.java
```

```
javac -cp .:gson-2.10.jar -d ./bin/ -sourcepath ./client/:./condivisi/
./client/ClientMain.java
```

Eseguire per primo il Server con:

```
java -cp .:gson-2.10.jar:./bin/ ServerMain
```

Aprire un secondo terminale, spostarsi sulla cartella GinevraMaoroWordle ed **eseguire** il client con:

```
java -cp .:gson-2.10.jar:./bin/ ClientMain
```

Su Windows:
Aprire il terminale e posizionarsi all'interno della cartella **GinevraMaoroWordle**, a questo punto **compilare** Server e Client rispettivamente con i comandi

```
javac -cp gson-2.10.jar -d .\bin\ -sourcepath ".\server\;.\condivisi\"
.\server\ServerMain.java
```

```
javac -cp gson-2.10.jar -d .\bin\ -sourcepath ".\client\;.\condivisi\"
.\client\ClientMain.java
```

Eseguire per primo il Server con:

```
java -cp gson-2.10.jar;.\bin ServerMain
```

Aprire un secondo terminale, spostarsi sulla cartella **GinevraMaoroWordle** ed **eseguire** il client con:

```
java -cp gson-2.10.jar;.\bin ClientMain
```

Introduzione

Il progetto consiste nella implementazione di WORDLE, un gioco nel quale l'utente deve trovare una parola inglese formata da 10 lettere, impiegando un numero massimo di 12 tentativi. WORDLE dispone di un vocabolario di parole di 10 lettere (*words.txt*), da cui estrae casualmente ogni 60s (60000 ms) una nuova Secret Word che sarà quella che gli utenti dovranno indovinare fino alla successiva estrazione. Fornisce anche un sistema di notifica di aggiornamento delle prime tre posizioni della classifica è la possibilità di condividere una partita giocata su un gruppo multicast.

Il gioco è implementato mediante due componenti principali: il WordleClient e il WordleServer. Il primo gestisce l'interazione con l'utente tramite CLI e comunica con i WordleServer.
Successivamente al login dell'utente viene instaurata una connessione TCP persistente con la quale si ha l'interazione del client col server attraverso l'invio dei comandi da parte del client e le risposte da parte del server. Per la comunicazione tra client e server sulla connessione TCP è stato utilizzato il multiplexing dei canali mediante NIO, con l'utilizzo di un threadpool da parte del server per la gestione delle richieste.

Istruzioni sulla sintassi dei comandi per eseguire le varie operazioni su Wordle

L'utente interagisce col client di Wordle attraverso una Comand Line Interface. All'avvio del client le prime due operazioni concesse all'utente sono quella della registrazione o del login.
Se l'utente è già registrato può eseguire il comando **login** ed inserendo il proprio username e password sarà in grado di accedere a Wordle. Altrimenti l'utente dovrà registrarsi attraverso il comando **reg** e in seguito eseguire il login. Registrazione e Login sono state realizzate tramite RMI.

```
**************************************************************
WORDLE: un gioco di parole 3.0
**************************************************************
Vuoi registrarti o effettuare il login?
!Ricorda!: non puoi effettuare il login se non sei registrato
Digita:
login -> per effettuare il login
reg -> per effettuare la registrazione
login
Inserisci il tuo username
ginevra
Inserisci la tua password
maoro
ok:connessione andata a buon fine
```

una volta avvenuto con successo il login:

1. Il client si registra ad un servizio di notifica del server per ricevere aggiornamenti sulle prime tre posizioni della classifica degli utenti, servizio implementato con il meccanismo di RMI callback.

2. Il client si unisce ad un gruppo multicast dal quale riceve mediante messaggi UDP, inviati dal server, l'esito delle partite giocate (vinte, perse o rinunciate) dagli altri utenti che hanno deciso di condividerle.

3. L'utente potrà eseguire uno dei seguenti comandi: gioca, stat, mc, class, logout

```
Benvenuto su Wordle ginevra!
puoi digitare uno dei seguenti comandi:
gioca-> per iniziare una nuova partita
stat-> per visualizzare le tue statistiche di gioco
mc-> (mostra condivisioni) per visualizzare le condivisioni dei risultati degli altri giocatori
class-> per visualizzare la classifica
logout-> per uscire dal gioco
```

Se l'utente ha deciso di giocare deve digitare **gioca** avviando così una partita, almeno che non abbia già giocato la parola del giorno.

```
gioca
errore hai già giocato la parola del giorno

puoi digitare uno dei seguenti comandi:
gioca-> per iniziare una nuova partita
stat-> per visualizzare le tue statistiche di gioco
mc-> (mostra condivisioni) per visualizzare le condivisioni dei risultati degli
altri giocatori
class-> per visualizzare la classifica
logout-> per uscire dal gioco
```

```
gioca
ok puoi giocare

Hai avviato una partita
puoi digitare uno dei seguenti comandi:
send ^guess word^-> per provare a indovinare la parola
exit-> per abbandonare la partita
```

La partita sarà relativa alla Secret Word di quel momento, se questa cambia mentre la partita è ancora in corso l'utente può continuare la partita relativa alla vecchia Secret Word.
Una volta avviata una partita l'utente può inviare un tentativo per indovinare la Secret Word col comando **send ^guess word^**. Se la Guessed Word non è contenuta nel vocabolario, questo tentativo non verrà contato tra i 12 a disposizione e verrà richiesto di inviare una nuova Guessed Word.

```
send xxxxxxxxxx

La parola che hai inviato non è presente nel vocabolario
prova con un'altra parola
ti sono rimasti 12 tentativi
```

Se la Guessed Word è più breve o più lunga di 10 caratteri, questo tentativo non verrà contato tra i 12 a disposizione e verrà richiesto di inviare una nuova Secret Word.

```
send 1234567891011

^guess word^ troppo breve/lunga, deve essere di 10 caratteri
```

Se la Guessed Word non coincide con la Secret Word, verranno diminuiti di 1 i tentativi, verrà mostrato il suggerimento e varra chiesto di ritentare di indovinare la Secret Word.

```
send papaphobia

Non hai indovinato, questo è il suggerimento
?x?x??x?+x
ti sono rimasti 11 tentativi
```

Nel suggerimento il simbolo "?" significa che la lettera corrispondente a quel simbolo nella Guessed Word è presente nella Secret Word ma non è nella giusta posizione.
Il simbolo "x" significa che la lettera corrispondente a quel simbolo nella Guessed Word non è presente nella Secret Word.

Il simbolo “+” significa che la lettera corrispondente a quel simbolo nella Guessed Word è presente nella Secret Word ed è anche nella giusta posizione.

Se l’utente invece riesce ad indovinare la Secret Word, verrà mostrata la traduzione e verrà chiesto se si vuole condividere il proprio risultato sul gruppo multicast, in caso affermativo l’utente dovrà digitare “SI” altrimenti “NO”.

```
send membership
HAI VINTO!

la parola da indovinare era membership
la sua traduzione è "abbonamento"
Vuoi condividere i risultati della partita appena giocata?

Digita SI per condividere, NO altrimenti

SI
ok condivisione andata a buon fine
```

Se l’utente non riesce a indovinare la parola, con i 12 tentativi a disposizione, la partita è considerata persa ma è comunque possibile condividere il risultato sul gruppo multicast.

```
send neurotonic
Hai finito i tentativi

la parola da indovinare era neogenetic
la sua traduzione è "neogenetico!"
Vuoi condividere i risultati della partita appena giocata?

Digita SI per condividere, NO altrimenti

SI
ok condivisione andata a buon fine
```

Nel caso in cui invece l’utente durante una partita digita il comando **exit**, la partita viene considerata come abbandonata (non vinta), e la parola del giorno considerata come giocata. È comunque possibile condividere il risultato sul gruppo multicast.

```
exit

la parola da indovinare era peninsular
la sua traduzione è "peninsulare!"
Vuoi condividere i risultati della partita appena giocata?

Digita SI per condividere, NO altrimenti

SI
ok condivisione andata a buon fine
```

Una volta finita una partita si essa conclusa con successo, perdita o rinuncia, l’utente può continuare a interagire con il client, attraverso gli altri comandi.

Il Comando **stat** mostra le statistiche aggiornate dopo l'ultima partita.

```
puoi digitare uno dei seguenti comandi:
gioca-> per iniziare una nuova partita
stat-> per visualizzare le tue statistiche di gioco
mc-> (mostra condivisioni) per visualizzare le condivisioni dei risultati degli
altri giocatori
class-> per visualizzare la classifica
logout-> per uscire dal gioco
stat
Le tue statistiche
N partite giocate: 5
% partite vinte: 60.0%
Streak: 1
MAX Streak: 2
Guess Distribution:
1 tentativo: 1
2 tentativi: 1
3 tentativi: 0
4 tentativi: 1
5 tentativi: 0
6 tentativi: 0
7 tentativi: 0
8 tentativi: 0
9 tentativi: 0
10 tentativi: 0
11 tentativi: 0
12 tentativi: 0
Punteggio: 1.5
```

Il Comando **class** mostra l'intera classifica, mostrando nome utente e punteggio di ogni utente

```
puoi digitare uno dei seguenti comandi:
gioca-> per iniziare una nuova partita
stat-> per visualizzare le tue statistiche di gioco
mc-> (mostra condivisioni) per visualizzare le condivisioni dei risultati degli
altri giocatori
class-> per visualizzare la classifica
logout-> per uscire dal gioco
class
CLASSIFICA:
1° manuele 12.0
2° ginevra 1.5
3° sandra 1.0
4° alle 0.0
```

Il comando **logout** esegue il logout dell'utente da Wordle con conseguente chiusura della connessione TCP, chiusura dello scanner per leggere i comandi da linea di comando, deregistrazione dal servizio di notifica di aggiornamento delle prime tre posizioni della classifica, chiusura del multicast socket per la ricezione delle condivisioni delle altre partite giocate dagli altri utenti e terminazione del client.

```
logout
ok
Client: chiudo il socket channel
Client: chiudo lo scanner
Client: mi deregistro dal servizio di notifica
Client: chiudo il multicast socket
Client: chiusura in finally
```

Il comando **mc** mostra tutte le condivisioni ricevute dal gruppo multicast, inviate dal server per conto degli alti client. Tutte le condivisioni vengono mostrate, anche le proprie, come se fosse una bacheca. Una volta però visualizzate vengono cancellate.

```
mc
ginevra
Wordle 25 2/12
?x?x??x?+x
++++++++++

ginevra
Wordle 60 0/12

ginevra
Wordle 73 12/12
++xx????++
++xx????++
++xx????++
++xx????++
++xx????++
++xx????++
++xx????++
++xx????++
++xx????++
++xx????++
++xx????++

alle
Wordle 106 4/12
???x???xx?
??x??x?xx?
??x??x?xx?
++++++++++

sandra
Wordle 107 4/12
x?xx++?xx?
x?xx++?xx?
x???xxx?xx
++++++++++

manuele
Wordle 107 1/12
++++++++++
```

Esempio in cui l'utente ha avviato una partita ma l'ha subito abbandonata con exit

Esempio in cui l'utente ha avviato una partita ma ha esaurito tutti i tentativi senza indovinare la parola

Esempio in cui l'utente ha indovinato la parola in 4 tentativi

Esempio in cui l'utente ha indovinato la parola al primo tentativo

Il formato della condivisione è composto da: nome utente, numero della parola estratta da Wordle, tentativi effettuati su 12, suggerimenti ottenuti con stringa finale "++++++++++" solo se è stata indovinata la parola.

Mentre l'utente è loggato può ricevere notifiche di aggiornamento delle prime tre posizioni della classifica in qualsiasi momento, la notifica avrà il seguente aspetto

```
Ricevuta notifica di aggiornamento della top 3
1° manuele 13.0
2° ginevra 1.5
3° sandra 1.0
```

Se nel momento del login, il server si rende conto che l'utente è già loggato e connesso su un altro client, fornisce la possibilità di disconnettersi per connettersi col nuovo client. Questo perché un utente può essere loggato solo su un client alla volta.

```
*****************************************************************
WORDLE: un gioco di parole 3.0
*****************************************************************
Vuoi registrarti o effettuare il login?
!Ricorda!: non puoi effettuare il login se non sei registrato
Digita:
login -> per effettuare il login
reg -> per effettuare la registrazione
login
Inserisci il tuo username
ginevra
Inserisci la tua password
maoro
questo utente è già loggato
se vuoi effettuare la disconnessione dall'altro dispositivo digita OK
DISCONNETTENDOTI perderai la partita in corso sull'altro dispositivo
comandi disponibili:
OK -> PER DISCONNETTERTI
NO -> PER NON disconnetterti
OK
ok:disconnessione e riconnessione andata a buon fine
```

Panoramica del Server con le sue strutture dati e thread attivati

Il Server di Wordle ha il ruolo di rappresentare il cuore del gioco, si occupa della gestione del database delle registrazioni che contiene tutte le informazioni e le statistiche degli utenti registrati. Tiene traccia di una classifica, ha il compito dell'estrazione periodica della nuova Secret Word, gestisce le partite dei vari client, fornendo i suggerimenti e la traduzione della Secret Word. Si occupa anche del servizio di notifica di aggiornamento delle prime tre posizioni della classifica e, su richiesta dell'utente, condivide i risultati di una partita sul gruppo multicast.

Strutture dati e thread:

- Thread pool `service` di tipo FixedThreadPoll con numero fisso di thread uguale al numero di core della macchina. Al thread pool verranno passati i vari task per gestire le richieste ricevute dai client. È stato scelto un FixedThreadPoll per abbattere i costi della creazione ed eliminazione dei thread e per evitare un consumo eccessivo delle risorse del sistema dato dall'alto numero di thread attivi, visto che nel FixedThreadPoll i thread sono fissati. Con lo svantaggio però di un possibile rallentamento delle prestazioni se il carico di lavoro è molto alto, cioè ci sono tante richieste da parte dei client e quindi vengono sottomessi tanti task al FixedThreadPoll.

- L'oggetto remoto per il servizio di notifica di aggiornamento della classifica `notify_server`, istanza di **`Server_NotifyEvent_impl`**.

- L'oggetto remoto `registro` per il servizio di registrazione, istanza di **`registro_registrazioni_impl`**. Nel momento in cui viene creato questo oggetto viene creato anche il registro (`Registro`) con tutti gli utenti registrati,

andando a deserializzare il file json “Registro_registrazioni.json” che contiene gli utenti registrati a Wordle con tutte le loro informazioni.

- la classifica (`classifica`) come istanza della classe **`Classifica`**.

- thread `serializ_registro` che si occupa della serializzazione periodica della struttura dati `Registro` all’interno del file “Registro_registrazioni.json”, la frequenza di serializzazione è uno di quei paramenti presenti nel “Configurazione_Server.properties”.

- Un hashSet `parole` che contiene tutte le parole del vocabolario di Wordle, viene creato andando a estrarre queste parole dal file “word.txt”.

- Un oggetto `SWO` di tipo **`SecretWordObject`** che serve a mantenere la Secret Word corrente, il suo numero, la sua traduzione e tutti gli utenti che l’hanno già giocata/stanno giocando.

- Thread `estrattore_sw` che si occupa di estrarre la nuova Secret Word e di modificare opportunamente in maniera thread safe l’oggetto `SWO`.

- Una struttura dati `partite` che è una ConcurrentHashMap che contiene l’ultima partita giocata da ogni utente.

- Una struttura dati `connessi` che è una hashMap che memorizza, per ogni connessione con i client, il SocketChannel lato server di quella connessione.

Il server è ora pronto per ricevere le connessioni da parte dei client, comunica col client tramite Java new IO in modalità non bloccante, cioè utilizzando canali non bloccanti associati ai socket TCP, per permettere così il multiplexing dei canali. Grazie al multiplexing avrò un selettore che esamina i NIO channels per determinare quali sono quelli pronti per operazioni di rete, da essi riceverà i comandi provenienti dai relativi client e di conseguenza passerà un task per soddisfare e rispondere a quel determinato comando ad uno dei thread del threadpoll. I vari task verranno eseguiti in maniera parallela all’interno del threadpool, ogni thread nell’esecuzione del task accederà concorretemene a strutture dati condivise; quindi, saranno utilizzati opportuni meccanismi di sincronizzazione. Una volta che il task avrà soddisfatto la richiesta del client risponderà ad esso attraverso il corrispettivo SocketChannel.

Panoramica del Client con le sue strutture dati e thread attivati

Il Client di Wordle ha il ruolo di interfacciarsi con l’utente, offrendogli la possibilità di effettuare richieste al Server, il quale risponderà al Client che mostrerà, sul terminale, le risposte all’utente e le successive possibili operazioni che possono essere effettuate.

Il client si mantiene lo stub relativo all’oggetto remoto che si trova sul server e che offre il servizio di registrazione e login implementato mediante RMI. L’utente può essere loggato solo su un client, cioè non posso avere lo stesso utente loggato su due client diversi. Una volta effettuato con successo il login viene instaurata la connessione TCP persistente col server, sulla quale poi verranno scambiate le richieste e le risposte.

Il client si mantiene lo stub relativo all'oggetto remoto che implementa il servizio di notifica di aggiornamento della classifica, che si trova sul server, implementato con il meccanismo di RMI callback, e si registra ad esso attraverso l'invocazione di un metodo remoto.

Viene avviato un thread `gruppo_multicast` che fa si che il client si unisca al gruppo multicast, dal quale riceverà le notifiche di condivisione delle partite degli altri utenti. Il thread esegue un task **`Multicast_client`** che è sempre in attesa dei messaggi di notifica ricevuti dal gruppo multicast per andarli a memorizzare nella ConcurrentHashmap `notifiche_di_share`, in modo che poi l'utente li possa visualizzare in un secondo momento.

Collections.synchronizedList(new ArrayList<String>());

A questo punto l'utente può inoltrare varie tipologie di richieste al server. I metodi associati ai comandi si preoccuperanno di inviare richieste con una opportuna sintassi stabilita dal protocollo di comunicazione, attraverso la connessione persistente TCP e con l'utilizzò di channel bidirezionali in modalità bloccante che si appoggiano a ByteBvuffer. Sono stati scelti i channel per la loro caratteristica di essere bidirezionali e in modalità bloccante perché il client, dopo una richiesta, prima di proseguire nella sua esecuzione sequenziale, deve attendere la riposta dal server e a seconda della risposta ottenuta potrebbe proseguire in una certa direzione piuttosto che in un'altra.

Classi, interfacce e Strutture dati

1.ClientMain

La classe **`ClientMain`** contiene il metodo main dal quale parte l'esecuzione del client, in essa viene creato un FileInputStream che contiene il contenuto del file "Configurazione_Client.properties" che a sua volta contiene i parametri di configurazione del client. Il FileInputStream a questo punto viene utilizzato per creare un oggetto di tipo Proprierties dal quale verranno estratti i parametri di configurazione come:

- **`registry_port`**: la porta associata al registry che memorizza gli stub degli oggetti remoti.
- **`porta_socket_TCP`**: porta associate alla socket TCP del server.
- **`porta_callback`**: porta associata al servizio remoto di notifica di aggiornamento della classifica con callback.
- **`indirizzo_multicast`**: indirizzo IP del gruppo di multicast dal quale si ricevono le notifiche di condivisione delle partite.
- **`porta_multicast`**: porta associata al MulticastSocket dal quale ricevo le notifiche del gruppo di multicast.
- **`socket_tcp_timeout:`** timeout associato alla socket associata al SocketChannel bloccante attraverso il quale si comunica col server.

A questo punto viene creato un oggetto **`Client`** (`Wordle_client`), passandogli tutti i parametri di configurazione e poi verrà avviato attraverso l'invocazione del metodo avvio(), avviando cosi il client.

2.Client

Questa classe fornisce l'implementazione del client di Wordle, la sua esecuzione parte dal metodo avvio(). Per prima cosa il client ottiene un riferimento al Registry dal quale ottiene lo stub del servizio remoto di registrazione attraverso una operazione di lookup().

Inizialmente l'utente, su quel client, non sarà loggato, questa condizione è identificata dal fatto che la variabile `logged` è settata a false. Entro così in un ciclo while che continua la sua iterazione finché la variabile non sarà a true, cioè l'utente si sarà loggato con successo.
All'interno del while accetto come comandi possibili dall'utente, da leggere da linea di comando attraverso il metodo next() dello scanner, solo "reg" e "login".
In uno switch gestisco il caso in cui l'utente abbia digitato "login" o "reg":

- case "**`reg`**": vengono fatti inserire nome utente e password, controllando che la password non sia la stringa vuota in un while, per dare la possibilità di inserirla di nuovo. Prima di completare con successo la registrazione si chiama il metodo remoto **`add_registrazione`**, che restituirà true solo se quel nome utente non è già stato utilizzato, permettendo così di andare avanti, se quel nome utente è già stato utilizzato restituirà false permettendo però grazie all'utilizzo di un while di reinserire il nome utente.

- case "**`login`**": vengono fatti inserire nome utente e password, attraverso l'invocazione dei metodi remoti dell'oggetto remoto, che implementa il servizio di registrazione, si verifica se il nome utente è presente nel registro e se la password è corretta. A questo punto verifico, sempre chiamando un metodo remoto, se quell'utente è già loggato su qualche altro client, consultando una variabile `logged` all'interno dell'oggetto **`Utente`** che rappresenta l'utente. Se non lo è lo logga, col metodo remoto setlogged, e invia un messaggio attraverso la SocketChannel al Server del tipo "nomeutente connesso" e si mette in attesa di una risposta, la quale arriverà una volta che il server avrà elaborato la richiesta. Adesso la variabile `logged` nel client verrà settata a true e quindi si uscirà dal while che aveva come guardia proprio questa variabile. Se invece dalla chiamata del metodo remoto mi rendo conto che l'utente è già loggato, per esempio su un altro dispositivo/client, do l'opportunità all'utente, digitando "DISCONNETTI" di effettuare il logout, e quindi la disconnessione sul vecchio client, e di loggarsi e quindi connettersi al server con l'attuale client. Può risultare che l'utente sia già loggato anche nel caso in cui, nel vecchio client, abbia avuto una chiusura improvvisa (es col comando Ctrl-C) e quindi non sia riuscito a mettere in atto la giusta routine di logout che prevede il settaggio della variabile `logged` dell'utente a false. Nel caso l'utente digiti "DISCONNETTI" viene inviato al server un messaggio del tipo "nomeutente DISCONNETTI" che verrà elaborato dal server, il quale risponderà al client in attesa, anche in questo caso si uscirà dal while settando la variabile "`logged`" a true.

Una volta avvenuto il login, il client si registra a un servizio di notifica implementato con RMI e call back: si reperisce lo stub `server` dell'oggetto remoto **`Server_NotifyEvent_impl`** del server per il servizio di notifica attraverso una lookup nel registry. Crea il proprio oggetto remoto `callbackObj`,di tipo

NotifyEvent_impl, che conterrà il metodo che il server invocherà per notificarmi, esporta l'oggetto remoto del client ottenendo uno stub `stub_notify`, infine registra lo `stub_notify` col metodo remoto **registerForCallback** sul `server` per ricevere le callback.

A questo punto si unisce al gruppo multicast sul quale il server invia le notifiche di share, per conto degli altri client che hanno manifestato la loro volontà di condividere i risultati delle loro partite. Per fare ciò si crea un thread `gruppo_multicast` che esegue la Runnable **Multicast_client**. Con la creazione di un nuovo oggetto **Multicast_client**, a partire dal metodo costruttore, viene inizializzata la struttura dati `notifiche_di_share` come una ArrayList sincronizzato attraverso il metodo SynchronizedList della classe Collections. E' stata fatta questa scelta perché può succedere che il thread `gruppo_multicast` aggiunga elementi ArrayList mentre il thread del main del client li vada a prelevare col metodo **showMesharing**, ma essendo ArrayList sincronizzato, non avrò problemi di concorrenza. Dopodiché sempre nel costruttore della classe **Multicast_client** viene aperto un MulticastSocket, sulla porta opportuna, e mi unisco al multicast group attraverso il metodo joinGroup(), in modo che dal MulticastSocket si possano ricevere i messaggi che provengono dal gruppo di multicast. Quando si eseguirà la start del thread `gruppo_multicast` viene eseguito il metodo run() del **Multicast_client** che riceverà pacchetti dal MulticastSocket che contengono le condivisioni delle partite, che vengono inseriti, come stringhe, nella struttura dati `notifiche_di_share` in modo che il client le possa consultare in un secondo momento. Il thread `gruppo_multicast` è sempre in attesa di ricevere pacchetti dal MulticastSocket.

Adesso il client è pronto per ricevere gli altri comandi dall'utente, si ha un while che ha come guardia la variabile `continua`, finché `continua` è a true il client continua a leggere i comandi inseriti dall'utente. Una volta letto il comando si sviluppa uno switch case per capire quale comando è stato inserito e come agire di conseguenza. I comandi disponibili in questo punto sono:

- **logout**: che consiste nell'invio del messaggio "nomeutente logout" al server con il metodo **Scrittura_nel_canale**. Questo metodo usa un primo ByteBuffer per inserirci la lunghezza del messaggio da inviare, scrive questa lunghezza sul SocketChannel leggendola dal ByteBuffer. Poi usa un secondo ByteBuffer, che contiene il messaggio da inviare, per leggere il messaggio da esso e scriverlo nel SocketChannel. Si utilizza questo metodo così il server sa che per prima cosa riceverà la lunghezza del messaggio, dalla quale capisce poi quanti sono i successivi Byte da leggere dal SocketChannel per leggere l'intero messaggio. Il client dopodiché si blocca in attesa di una risposta che verrà letta dal SocketChannel e scritta in un ByteBuffer per poi essere mostrata all'utente sul terminale. A questo punto la variabile `continua` viene settata a false per far si che si esca dal while. Nella clausola finnaly del try-catch ci si occuperà di chiudere il SocketChannel, gli scanner, il MulticastSocket, deregistarsi dal servizio di notifica e chiudere il client.
- **gioca**: che consiste nella richiesta di avviare una nuova partita relativa all'attuale Secret Word. Per prima cosa vengono allocate due variabili `traduzione` e `secretword`. Per gestire la richiesta di gioco viene chiamato il metodo **playWordleCLIENT**. Questo come prima cosa, chiamando il

metodo **`Scrittura_nel_canale`**, descritto prima, invia al server il messaggio "nomeutente gioca", successivamente si alloca un ByteBuffer per leggere la risposta. Se andando a leggere dal SocketChennel con la read() si ottiene come valore di ritorno il valore -1, vuol dire che si è raggiunto l'end of streams quindi, la Socket remota lato server è stata chiusa. Questo può avvenire per esempio nel caso in cui l'utente ha espresso la volontà di loggarsi su un secondo client, quando però era già loggato su un altro, il server ha ricevuto quindi il messaggio "nomeutente DISCONETTI" ed è andato a chiudere la sua connessione col vecchio client. In questo caso il metodo **`playWordleCLIENT`**, eseguito sul vecchio client, ritorna -1 facendo si che la variabile `continua`, che fa da guardia del while, che permette la sottomissione da parte dell'utente dei comandi al client, venga impostata a false, impedendo la ricezione di altri comandi e portando alla chiusura del client. Se invece attraverso la read() si è riusciti a leggere la risposta dal server, si possono essere ottenute due tipi di risposta

1. "errore:hai già giocato la parola del giorno" con la quale **`playWordleCLIENT`** ritorna 0 e non da la possibilità di giocare una partita, ma l'utente può continuare a sottomettere comandi al client
2. "ok:puoi giocare:"+secretword+":"+traduzione" con la quale **`playWordleCLIENT`** ritorna 1 e setta i valori della `secretword` e della sua `traduzione` nel client. Viene così avviata una partita, si entra in un while che ha come guardia la variabile `ok` che rimane settata a true almeno che l'utente non digiti **`exit`**, per abbandonare la partita, o se vengono finiti i tentativi per indovinare la parola. In questo while i due comandi possibili sono **`send`** o **`exit`**. Se l'utente digita **`send`** insieme al comando deve aver inserito anche la Guessed Word, cioè la parola con la quale tenta di indovinare la Secret Word. Se la Guessed Word non è di dieci caratteri viene fatto reinserire uno dei due comandi send o exit, altrimenti se ha la giusta lunghezza viene eseguito il metodo **`sendCLIENT`**. Con esso si invia il messaggio "nomeutente + " " + "tentativo"+ " " +guessWord" e ci si mette in attesa di una risposta dal server, come nel caso del metodo **`playWordleCLIENT`** si può ottenere come ritorno dalla read() il valore -1 (per lo stesso motivo descritto nel metodo **`playWordleCLIENT`**). Questo fa si che il metodo **`sendCLIENT`** ritorni -1 portando all'uscita dei ciclo while per i comandi relativi alla partita (send e exit) e all'uscita del while superiore con conseguente chiusura del client e opportuna chiusura dei vari socket, scanner ecc nel finally.
   Se **`sendCLIENT`** ha invece ottenuto la risposta dal server, controlla se ha ricevuto il messaggio che indica la vincita (tentativi_rimasti+":vinto") ritornando 1 se è questo il caso. Altrimenti controlla quanti tentativi sono rimasti all'utente, estraendoli dalla stringa di risposta del server, se sono arrivati a zero, stampa a terminale che sono finiti i tentativi e invia il messaggio "nomeutente + " " + "exit" al server, il quale effettuerà le opportune operazioni come aggiornare le statistiche dell'utente in quanto questa partita è considerata come persa. Ritorna 1 anche in questo caso.

Un'altra risposta che può ottenere dal server è un messaggio che contiene la parola “suggerimeto” (tentativi_rimasti+":"+"suggerimento:"+suggerimento effettivo), in questo caso non si è indovinato la Secret Word ma si hanno ancora tentativi, e ci viene mostrato il suggerimento relativo alla Guessed Word appena inviata. In questo caso si ritorna 0.
Infine, l’ultima risposta che può ottenere dal server è quella che contiene la stringa “errore2” (tentativi_rimasti+":"+"errore2") che mi indica che la parola inviata non è contenuta nel vocabolario di Wordle, e che quindi, non mi sono stati diminuiti i tentativi e posso continuare a indovinare la secret word. In questo caso ritorna 0.
Quando **`sendCLIENT`** ritorna 0, posso ancora tentare di indovinare la parola, quando ritorna 1 significa che ho finito i tentativi o che ho indovinato la Secret Word, quindi esco dal while che mi permetteva di inviare i comandi send e exit, settando la guardia, cioè la variabile `ok`, a false. L’altro comando che a questo punto poteva digitare l’utente è il comando **`exit`** col quale si abbandona la partita andando a scrivere sul canale il messaggio “nomeutente exit” e si setta la guardia del while, la variabile `ok`, a false.

Successivamente, sempre nel case “**`gioca`**” del client, nel caso si sia potuto giocare una partita con successiva perdita, vincita o abbandono, si avrà la variabile `avanti` settata a true, che ci permetterà di entrare in un while nel quale si chiede all’utente se vuole condividere sul gruppo multicast il risultato di quella partita. Se l’utente digita “OK”, il client invia al server il messaggio “nomeutente + " " + "share"” che una volta ricevuto dal server farà si che quest’ultimo condivida la partita di questo utente sul gruppo multicast e risponda “ok” al client che quindi andrà avanti nella sua esecuzione dando la possibilità all’utente di effettuare altri comandi.

- **`stat`**: Comando col quale l’utente richiede che gli vengano mostrate le proprie statistiche. Viene invocato il metodo **`statistics`** che invia il messaggio “nomeutente + " " + "statistics"” al server, il quale ci risponderà con le statistiche di quell’utente, che verranno quindi lette dal SocketChennel con una read(), inserite in un ByteBuffer dal qual poi verranno spostate in una stringa che verrà stampata a video. Anche qui nel caso in cui effettuando la read() sul SocketChennel si trovi la socket remota chiusa, **`statistics`** ritornerà -1 con conseguente chiusura del client come già spiegato precedentemente anche per gli altri metodi.

- **`class`**: Comando col quale l’utente richiede di visualizzare la classifica. Viene invocato il metodo **`ShowMeRanking`** che invia il messaggio “nomeutente + " " + "classifica"” al server, il quale ci risponderà con l’intera classifica, che verrà letta dal SocketChennel con una read(), inserita in un ByteBuffer dal quale poi verra spostata in una stringa che verrà stampata a video. Anche qui nel caso in cui effettuando la read() sul SocketChennel si trovi la socket remota chiusa, **`ShowMeRanking`** ritornerà -1 con conseguente chiusura del client come già spiegato precedentemente anche per gli altri metodi.

- **mc**: Con questo comando si richiede di visualizzare tutte le condivisioni/share delle partite degli altri utenti sul gruppo multicast, ricevute da quando l'utente è loggato. Viene chiamato il metodo **showMesharing**, che va a controllare come prima cosa se la struttura dati `notifiche_di_share`, nella quale il thread `gruppo_multicast`, eseguendo la runnable **Multicast_client** inserisce le condivisioni ricevute, è vuota; in questo caso viene stampato un messaggio che avverte di questa condizione e che quindi non sono state ricevute condivisioni. Se invece la struttura dati non è vuota se ne stabilisce la dimensione col metodo size() e con un for si vanno a estrarre ed eliminare, dalla testa della struttura dati, un numero di notifiche pari a size. Non è stato utilizzato un iteratore perché avrebbe potuto sollevarmi ConcurentModificationException, visto che il thread `gruppo_multicast` aggiunge in mutua esclusione elementi.

3.Utente

Questa classe mi permette di istanziare oggetti **Utente** che rappresentano gli utenti registrati a Wordle. Ogni utente ha un username univoco, una password, una variabile `logged` che mi indica se quell'utente è loggato su qualche client e un insieme di statistiche.
Le statistiche prevedono:

- `Npartite_giocate`: il numero di partite giocate (vinte,perse o abbandonate).
- `Percentuale_vinte`: la percentuale di partite vinte (partite giocate/partite vinte).
- `streak`: la lunghezza dell'ultima sequenza continua di vincite.
- `max_streak`: la lunghezza della massima sequenza continua di vincite.
- `guess_distribution`: rappresentata con un array che mi indica la distribuzione dei tentativi impiegati per arrivare alla soluzione del gioco in ogni partita vinta. Per esempio, all'indice 3 dell'array troverò il numero di partite che l'utente ha vinto con 4 tentativi (perché l'array parte da zero ma noi contiamo i tentativi da 1)
- `punteggio`: utilizzato per inserire l'utente in classifica, viene calcolato come: n° partite vinte*(1/numero tentativi medi per raggiungere la soluzione), in modo che il secondo fattore, con l'aumento dei tentativi medi vada a penalizzare il n° di partite vinte.

Nel momento della registrazione, in cui un nuovo utente viene creato, col suo username, la sua password, la variabile `logged` a false e tutte queste statistiche verranno settate a 0.

La classe mette a disposizione i metodi setter e getter per accedere e modificare i valori delle variabili private di ogni oggetto Utente. Tutti i metodi setter e getter sono sincronizzati col modificatore synchronized, che garantisce un accesso in mutua esclusione all'oggetto utente, poiché un utente potrebbe essere una risorsa condivisa da più thread e quindi di si potrebbe andare in contro a race condition. Gli unici metodi non synchronized sono quelli che accedono al nome utente e alla password, poiché queste informazioni sono immutabili, e i metodi setter e getter sulla variabile logged poiché vengono sempre usati all'interno di blocchi synchronized sull'intero utente.

4. Server_NotifyEvent_impl

Questa classe serve per implementare il servizio di notifica di aggiornamento delle prime tre posizioni della classifica, attraverso il meccanismo di RMI callback. Aggiornamento inteso sia che cambino i nomi utenti della top 3, sia che il punteggio di uno degli utenti nella top 3 cambi. La classe implementa l'interfaccia **Server_NotifyEvent_interface** la quale estende l'interfaccia Java.rmi.Remote e definisce i metodi remoti per registrarsi e deregistrarsi dal servizio di notifica. Inoltre, **Server_NotifyEvent_impl** estende RemoteObject poiché questa classe implementa oggetti remoti che contengono metodi che non sono definiti nell'interfaccia **Server_NotifyEvent_interface**, ma che devono poter essere chiamati da remoto (come il metodo **update**). All'interno della classe si definisce la variabile ArrayList `<NotifyEvent_interface> clients` che conterrà la lista degli stub dei clients registrati al servizio di notifica. Poi viene implementato il metodo **registerForCallback** dell'interfaccia, che verrà invocato dal client, da remoto, per registrarsi, questo metodo prende in input il riferimento all'oggetto remoto **NotifyEvent_interface** del client che conterrà, a sua volta, il metodo remoto (**notifyEvent**) che il server invocherà per notificarlo.
Viene anche implementato il metodo **unregisterForCallback**, anch'esso definito nell'interfaccia, che serve a effettuare la de registrazione dal servizio rimuovendo lo stub di quel client dall'ArrayList `clients`. Infine, col metodo **update** si chiama il metodo **DoCallbacks** che prende come parametri di input 3 stringhe che rappresentano il primo, il secondo e il terzo utente in classifica con il rispettivo nome utente e punteggio. All'interno di **DoCallbacks** con l'utilizzo di un ciclo for scandisco tutti gli stub dei client registrati al servizio di notifica e attraverso essi invoco il metodo remoto **notifyEvent**, che verrà eseguito sul client, che stamperà la notifica di aggiornamento della classifica. All'interno del for si gestisce anche il caso in cui si provi a invocare il metodo remoto **notifyEvent** su un client che non è più attivo, e col quale non si ha più una connessione, perché per esempio è stato vittima di una disconnessione improvvisa (es Ctrl-C) e non ha avuto modo di deregistrarsi dal servizio di notifica attraverso il metodo **unregisterForCallback.** Per gestire questa situazione si rimuove lo stub di questo client da `clients` e si ricalcola la dimensione dell'Arraylist `clients` su cui è in opera il ciclo for e si va avanti notificando il resto dei client. Tutti i metodi di questa classe hanno il modificatore synchronized poiché l'Arraylist `clients` è una struttura dati a cuoi possono accedere concorretemene più thread per andare a registrare e deregistrare gli sub dei vari client, quindi il loro accesso deve avvenire in mutua esclusione per evitare race condition.

5.entry_classifica

Questa classe serve appunto per rappresentare una entry nella classifica, entry costituita da nome utente e punteggio associato a quel nome utente. Ha due costruttori, uno prende come paramento di input solo il nome utente; viene utilizzato nel momento della registrazione quando devo aggiungere un utente in classifica e quindi la sua entry avrà un punteggio associato a 0. Il secondo costruttore serve invece ad aggiungere una entry aggiornata in classifica; che contiene il punteggio aggiornato passato in input,

dopo una partita dell'utente. Ha due metodi getter per le variabili private punteggio e nome utente, non possiede metodi setter perché se devo aggiornare il punteggio di una entry associata a un utente in classifica, non vado a modificare la vecchia entry, ma elimino la vecchia e ne creo una nuova, col nuovo punteggio, andandola a inserire nella giusta posizione della classifica a seconda del punteggio.
Questa classe implementa l'interfaccia Comparable, per permettere di confrontare oggetti di questa classe. Viene effettuato l'override del metodo equals() che mi restituisce true solo se due entry hanno lo stesso nome utente, sfruttando il metodo equals() tra stringhe . Si effettua l'override anche del metodo compareTo(), che mi permette di stabilire una relazione di ordinamento tra le entry della classifica in base al punteggio contenuto nelle entry, sfruttando il metodo compare() tra Double.

6.Classifica

La classe Classifica serve per rappresentare la classifica ordinata del gioco Wordle, è implementata con un ArrayList<**entry_classifica**>, essendo questa struttura dati non sincronizzata, tutti i metodi della classe Classifica hanno il modificatore synchronized, per garantire che non ci siano race condition, le quali si potrebbero verificare perché la classifica è una risorsa condivisa da più thread che possono accederle e modificarla in maniera contemporanea.
La classe contiene un metodo **add_entry_ordinata()** che viene chiamato sulla classifica nel momento in cui si avvia il server e quindi dobbiamo prendere tutti gli utenti dal registro e inserirli in maniera ordinata nella classifica, infatti al suo interno si crea una nuova entry con nome utente e punteggio, passati come paramenti di input del metodo, e si inserisce questa nuova entry nella giusta posizione, scorrendo la classifica e sfruttando il metodo compareTo() sulle **entry_classifica**.
Poi abbiamo il metodo **add_entry()** che viene usato per inserire in classifica un utente appena registrato e che quindi avrà punteggio 0. Infatti viene creata una nuova **entry_classifica** relativa a questo nome utente, e verrà inserita infondo alla classifica col metodo add(). Se la classifica ha ancora una dimensione inferiore a 3, anche se questo nuovo utente viene aggiunto infondo, finirà in una delle posizioni della top 3 e quindi in questo caso dobbiamo effettuare una callback per il servizio di notifica di aggiornamento della top 3, questo è possibile perché uno dei parametri di input dell'metodo è lo stub del servizio remoto di notifica **Server_NotifyEvent_impl**.
Invece il metodo **upgrade_classifica()** viene chiamato quando un utente vince una partita e quindi il suo punteggio cambia e di conseguenza potenzialmente anche la sua posizione in classifica. Perciò si crea una nuova **entry_classifica** con nome utente e il nuovo punteggio per quell'utente e grazie al metodo remove(), che sfrutta al suo interno il metodo equals() della classe **entry_classifica,** si rimuove la vecchia entry di quel nome utente, per poi andare a reinserire nella giusta posizione la nuova **entry_classifica** aggiornata, sfruttando in questo caso il metodo sovrascritto compareTo() della classe **entry_classifica**.
In seguito all'aggiornamento del punteggio dell'utente, la sua nuova entry aggiornata potrebbe essere inserita in una delle prime 3 posizioni della classifica, in questo caso viene fatta una callback per il servizio di notifica. Infine, in questa classe abbiamo un metodo per stampare l'intera classifica e un metodo per inserire l'intera classifica in una Stringa.

7.registro_registrazioni_impl

Con questa classe si vuole realizzare il servizio remoto di registrazione e login a Wordle. Essa implementa l'interfaccia **registro_registrazioni_interface** la quale estende l'interfaccia Java.rmi.Remote e contiene le definizioni dei metodi remoti. Partendo dal costruttore si va a leggere il file JSON "Registro_registrazioni.json", che contiene tutti gli utenti registrati e le loro informazioni, e lo si deserializza in un oggetto `ConcurrentHashMap <String, Utente> Registro` utilizzando la libreria GSON. È stata scelta questa tipologia di struttura dati perché posso estrarre un utente e tutte le sue informazioni in tempo costante, nelle HashMap non posso avere chiavi duplicate, ma questo non è un problema perché i nomi utente devono essere univoci. `Registro` può essere utilizzata da più thread contemporaneamente si è quindi scelto di usare una struttura sincronizzata per evitare di dover effettuare in maniera esplicita la sincronizzazione poiché le operazioni su diverse parti della mappa (dette segmenti) possono essere eseguite contemporaneamente dai thread senza interferire l'una con l'altra, infine fornisce dei metodi atomici come "putIfAbsent". Una volta creata la struttura dati `Registro`, setto la variabile `logged` di tutti gli utenti al suo interno a false perché all'avvio del server non avrò nessun utente loggato. Un metodo di questa classe è **add_registrazione** che permette di effettuare una registrazione a Wordle, andando a inserire un nuovo utente nel `Registro` e ritornando un booleano se questa operazione è stata possibile. In questo metodo si rende possibile l'aggiunta di un nuovo utente al registro solo se l'username dell'utente che si vuole registrare, passato come parametro di input del metodo, non è già contenuto nel `Registro`. Questo controllo viene fatto con l'operazione atomica putIfAbsent(), che appunto in maniera atomica controlla se quell'username non è presente nel registro e aggiunge il nuovo utente con quell'username, evitando problemi di concorrenza. Avendo appena registrato un nuovo utente questo viene anche aggiunto in fondo alla classifica attraverso il metodo **add_entry**. Poi è presente un metodo per stampare tutti gli utenti nel registro, metodi getter per ottenere l'utente associato ad un username, quello per ottenere la password di un utente nel registro e quello per ottenere un riferimento alla struttura dati Registro. Infine, abbiamo due metodi setter, **setlogout** che prende in input un nomeutente, va a ricavare l'oggetto utente associato a quel nomeutente nel registro e invoca il metodo **setlogout** della classe utente su quell'utente, che ricordiamo essere un metodo synchronized per evitare problemi di concorrenza. L'altro metodo setter è **setlogged**, che prende in input un nome utente, ricava l'oggetto utente associato a quel nome utente e all'interno di un blocco synchronized sull'oggetto utente vado a controllare se quell'utente è loggato con il metodo **getlogged** della classe **Utente**, e in caso non sia loggato lo loggo con il metodo **setlogged** della classe **Utente**, altrimenti comunico che l'utente è già loggato e ritorno false.

8. NotifyEvent_impl

Questa classe implementa l'interfaccia **NotifyEvent_interface** quindi fornisce una implementazione del metodo remoto **notifyEvent**, utilizzato per notificare il client di un aggiornamento delle prime 3 posizioni in classifica. All'interno di questa classe viene creato un ArrayList di stringhe che, quando il server da remoto chiamerà il metodo **notifyEvent**, passandogli come parametri di input 3 stringhe, saranno memorizzate nell'ArrayList e stampate lato client. Le tre stringhe rappresentano proprio la notifica di

aggiornamento della classifica, la prima sarà relativa al nome utente e al punteggio dell'utente in prima classifica e così via.

9. SecretWordObject

Con questa classe rappresento un oggetto che si memorizza al suo interno, in variabili private, la Secret Word "della giornata", la sua traduzione, il n°della Secret Word e una struttura dati HashSet `giocatori` che contiene i nomi utente che hanno già giocato quella Secret Word. Ho i metodi setter per cambiare la Secret Word, il metodo per settare il nuovo numero della nuova Secret Word estratta e il metodo per settare la nuova traduzione, queste modifiche vengono fatte quando estraggo la nuova Secret Word. Ho i metodi getter per leggere la Secret Word, la sua traduzione e il numero di Secret Word. Ho un metodo **`resetgiocatori`** che va a resettare la struttura dati `giocatori`, viene chiamato quando estraggo la nuova Secret Word e quindi nessun giocatore l'avrà ancora giocata.
Tutti questi metodi non sono sincronizzati perché li utilizzo nel main del server all'interno di blocchi synchronized sull'oggetto `SWO` istanza della classe **`SecretWordObject`**.

10. Partita

Questa classe mi serve per istanziare oggetti che rappresentano partite. Una partita è iniziata da un utente quando vuole provare a indovinare la Secret Word. Col costruttore si crea una partita che ha le caratteristiche di una appena avviata: avrà una Secret Word che è passata come stringa di input, che rappresenta la parola da indovinare in quella partita; la sua traduzione, il numero di Secret Word, poi avrò una variabile tentativi che inizialmente, quando l'utente non ha ancora inviato nessuna Guessed Word, è a 0. Conterrà una struttura dati `suggerimenti` LinkedList<String> che conterrà una lista di stringhe che rappresentano i suggerimenti relativi ad ogni Guessed Word sottomessa per indovinare la Secret Word. Infine, ha una variabile `flag` che serve a capire se la partita è stata già usata o meno per aggiornare le statistiche dell'utente che l'ha giocata, `flag` a 1 significa che non è stata usata per aggiornare le statistiche, 0 altrimenti. Contiene i relativi metodi setter e getter per ottenere e modificare le variabili di istanza e un metodo per ottenere una partita sotto forma di stringa.

11. ServerMain

La classe **`ServerMain`** contiene il metodo main dal quale parte l'esecuzione del Server, in essa viene creato un FileInputStream che contiene il contenuto del file "Configurazione_Server.properties" che a sua volta contiene i parametri di configurazione del server. Il FileInputStream a questo punto viene utilizzato per creare un oggetto di tipo Proprierties dal quale verranno estratti i parametri di configurazione come:

- **`porta_registro`**: la porta associata al Registry che memorizza gli stub degli oggetti remoti.

- **porta_notifica**: porta a cui associo il servizio di notifica di aggiornamento delle prime tre posizioni della classifica, implementato dell'oggetto remoto di tipo **Server_NotifyEvent_impl**.
- **porta_registrazioni**: porta a cui associo il servizio di registrazione e login, implementato dell'oggetto remoto di tipo **registro_registrazione_impl**.
- **frequenza_serializzazione**: frequenza in ms con la quale serializzato il `Registro` che contiene tutti gli utenti registrati e le loro informazioni, nel file "Registro_registrazioni.json", per rendere permanenti le sue informazioni lato server.
- **frequenza_aggiornamento_parola**: frequenza in ms con la quale viene cambiata la Secret Word da indovinare.
- **porta_socket_TCP**: porta associata alla socket TCP dal quale il server riceve le richieste di connessione.
- **indirizzo_multicast**: indirizzo IP del gruppo di multicast dal quale si ricevono le notifiche di condivisione delle partite.
- **porta_multicast**: porta associata al MulticastSocket dal quale ricevo le notifiche del gruppo di multicast.

A questo punto viene creato un oggetto **Server** `Wordle_server`, passandogli tutti i parametri di configurazione e poi verrà avviato attraverso l'invocazione del metodo avvio(), avviando cosi il Server.

12.Server

Questa classe fornisce l'implementazione del server di Wordle, nel momento in cui viene chiamato il suo metodo costruttore vengono passati tutti i parametri di configurazione e memorizzati in opportune variabili di istanza private, inoltre vengono istanziate anche alcune strutture dati e oggetti, tra cui:

- `partite`: una ConcurrentHashMap che usa come chiavi le stringhe che rappresentano i nomi utente e come valori oggetti che sono istanza della classe **Partita**. Questa struttura viene quindi utilizzata per tenere traccia delle ultime partite giocate da ogni utente. Quando un utente gioca una nuova partita, la vecchia partita di quell'utente, memorizzata nella struttura dati, viene sostituita dalla nuova. Essendo una ConcurrentHashMap le operazioni su `partite` sono thread-safe. Inoltre, la partita relativa a un nome utente in `partite` non può essere acceduta da più thread in contemporanea perché, le operazioni su di essa, le può "oridnare" solo un utente che è loggato su un client che manda richieste sequenziali, e non posso avere lo stesso utente loggato su client diversi.
- `connessi`: una HashMap che usa come chiavi le stringhe che rappresentano i nomi utente e come valori i SocketChannel che sono associati a delle connessioni con i client. Questa struttura viene utilizzata per memorizzare a quale utente è associata la connessione relativa a un SocketChannel, perché, come vedremo, posso avere la necessità di interrompere la connessione con un certo utente, andando a chiudere la SocketChannel lato server. Le operazioni di aggiunta o rimozione di elementi a questa struttura dati, per evitare race condition, vengono fatte tutte all'interno di blocchi synchronized su `connessi` perché più thread potrebbero andare a accedere in contemporanea a `questa` struttura dati.
- `classifica`: istanza della classe **Classifica**.

L'esecuzione parte dal metodo avvio(). Per prima cosa viene creato il fixedThreadPool con un numero di thread fisso, pari al numero di core della macchina. Successivamente creo un'istanza `notify_server` dell'oggetto remoto **`Server_NotifyEvent_impl`** che mi fornisce il servizio di notifica di aggiornamento della classifica con callback, ed esporto questa istanza ottenendone uno stub.
Creo il Registry nel quale pubblicare lo stub associandogli il nome "SERVIZIO-NOTIFICA". Adesso creo un'istanza `registro` dell'oggetto remoto **`registro_registrazioni_impl`** che mi implementa il servizio di registrazione e di login, lo esporto ricavandone lo stub che pubblicherò nel Registry col nome "SERVIZIO-REGISTRAZIONE". All'interno dell'istanza `registro` è contenuta una struttura dati `Registro` di tipo ConcurrentHashMap <String, Utente> che contiene associazioni tra nome utente e un oggetto di tipo Utente che contiene tutte le caratteristiche dell'utente identificato con quel nome utente. Col metodo getRegistro() ottengo il riferimento alla struttura dati `Registro`, dalla quale prima estraggo una collezione che contiene tutti i valori della ConcurrentHashMap, cioè tutti gli oggetti Utente; collezione che itero con un for per ricavare da ogni oggetto Utente il nome e il punteggio in modo da poterlo inserire all'interno di un oggetto che si chiama `classifica`, istanza della classe **`Classifica`**, che mantiene la classifica ordinata in base al punteggio degli utenti registrati a Wordle.
Adesso viene avviato un thread che si occupa di eseguire la Runnable **`serializzatore_Registro`**, che serializza con una certa frequenza la struttura dati `Registro`, contenuta nell'istanza `registro` dell'oggetto remoto, nel file "Registro_registrazioni.json" usando la libreria GSON.
Successivamente viene creata la struttura dati `parole`, come un HashSet, che rappresenterà il vocabolario di Wordle. E' stato scelto questo tipo di struttura dati perché l'operazione contains() su di essa è effettuata in tempo costante e la utilizzeremo per controllare se la parola inviata dall'utente per indovinare la Guessed Word è contenuta nel vocabolario di Wordle. Col metodo **`crea_set_parole`** verranno inserite tutte le parole dal file "words.txt" appunto proprio in questa struttura dati `parole`.
Adesso si crea un'istanza `SWO` della classe **`SecretWordObject`**, che mi conterrà la Secret Word, il numero di Secret Word estratta, la traduzione e una struttura dati che mi memorizza tutti gli utenti che hanno già giocato quella Secret Word. A questo punto avvio il thread che si occupa di eseguire la Runnable **`estrazione_nuova_parola`**, all'interno della quale dopo aver estratto un indice casuale, scorro tutte le parole nella struttura dati `parole` e quando arrivo a quella dell'indice casuale la seleziono come nuova Secret Word. Per ricavarne la traduzione accedo al servizio alla URL https://mymemory.translated.net/doc/spec.php tramite una chiamata HTTP GET. Ora devo aggiornare le informazioni all'interno dell'oggetto `SWO`, questo vien fatto in un blocco synchronized su `SWO`. Il thread dormirà per "frequenza_aggiornamento_parola" ms prima di estrarre la nuova Secret Word.

Il server ha concluso una prima fase di settaggio ed è pronto per ricevere le connessioni dai client. Come già detto la comunicazione tra client e server avviene tramite una connessione TCP e l'utilizzo del multiplexing dei canali mediante NIO. Quindi a questo punto viene aperto un ServerSocketChannel, reperisco il ServerSocket associato al channel per legarlo all'InetSocketAddress che rappresenta l'indirizzo IP del local host e la porta è quella che è stata presa dal file di configurazione del server. Setto adesso il ServerSocketChannel in modalità non bloccante per permettere il multiplexing dei

canali. Apro un selettore che mi permetterà di monitorare i NIO channels e intercettare eventi provenienti da essi, infine registro sul selettore il ServerSocketChannel con l'interesse di monitorare le operazioni di accept.

Entro in un while all'interno del quale, in maniera ciclica, effettuo una select() sul selettore, con la quale si crea un set di chiavi che rappresentano i canali, registrati sul selettore, che sono stati trovati pronti per quelle operazioni di I/O, di cui avevo espressamente segnalato la volontà di monitorare, quando li avevo registrati sul selettore. Questo insieme di chiavi viene estratto dal selettore con il metodo selectedKeys() e verrà rappresentato dalla variabile `readykeys`. Adesso tramite un iteratore si analizzano tutte le chiavi del set `readykeys`, una volta finite verrà effettuata un'altra select() sul selettore e cosi via. Tornando al momento in cui con l'iteratore si vanno ad estrarre le chiavi del `readykeys`, una volta estratta una chiave, questa viene rimossa dal `readykeys`. Per ogni chiave estratta si controlla, con key.isAcceptable(), se il canale associato a quella chiave è pronto ad accettare una nuova connessione, questo canale sarà sempre il ServerSocketChannel aperto inizialmente, che funge da welcome Socket, poiché sarà l'unico che registro sul selettore con la volontà di monitorare su di lui le operazioni di accept.
Se il canale è pronto per accettare una nuova connessione, lo estraggo dalla chiave a lui associata e attraverso il metodo accept() mi viene restituito un SocketChannel che mi collega col client dal quale ho accettato la connessione. Adesso questo nuovo SocketChannel, relativo alla nuova connessione con un client, lo imposto in modalità non bloccante; siccome da queste connessioni sono interessato solo a leggere le richieste, poiché le risposte verranno date dai thread del pool ai quali assegno gli opportuni task per rispondere in maniera corretta, andrò a registrare i socketChannel sul selettore con l'interesse di monitorare solo le operazioni di read, cioè il client ha scritto qualcosa sul canale, una richiesta, e quindi io server la devo leggere.
Nel momento in cui faccio la registrazione sul selettore passo anche un attachment che è un array di ByteBuffer che contiene due ByteBuffer. Il primo è un ByteBuffer con capacità sufficiente per memorizzare un intero, e servirà a leggere dal canale l'intero che il client ha scritto sul canale e che mi rappresenta la lunghezza del messaggio che il client mi sta inviando, come previsto dal protocollo. Il secondo sarà il ByteBuffer che conterrà il messaggio ricevuto. Si utilizza un attachment per accumulare i byte restituiti da una sequenza di letture non bloccanti.
Sulle chiavi che itero, il secondo controllo che faccio, è se il canale associato alla chiave in questione è pronto per essere letto, in caso affermativo estraggo il canale relativo alla chiave e recupero l'array di ByteBuffer dall'attachment della chiave con key.attachment(). A questo punto posso leggere dal SocketChannel e scrivere nei buffer dell'array di ByteBuffer, dopodiché si verifica se il buffer per la lunghezza del messaggio è stato riempito e quindi si è terminata la lettura dell'intero intero scritto sul canale. Se non è stato letto tutto si continuerà la lettura la prossima volta che verrà selezionata la chiave relativa a questo canale.
Se l'intero è stato completamente letto controllo che i byte nel secondo ByteBuffer, contenuto nell'array dell'attachment, siano pari alla lunghezza che ho ricevuto e scritto nel primo ByteBuffer, in caso negativo andrò avanti nella lettura del messaggio la prossima volta che estrarrò la chiave relativa a questo canale. In caso positivo creo una stringa dal secondo Bytebuffer dell'attachemt e questa stringa conterrà la richiesta ricevuta dal client. La prima parola del messaggio sarà il nome utente da cui ho ricevuto la richiesta, la seconda parte conterrà il comando del client come da protocollo. Con

uno switch determino il comando che ho ricevuto e passo a un thread del pool il task per rispondere a quel messaggio. I comandi che il server riconosce sono:

- *logout*: inviato dal client quando l'utente vuole effettuare il logout. Viene eseguita la runnable **`logout`**.
- *gioca*: il client lo invia quando vuole giocare, cioè vuole avviare una partita per indovinare la Secret Word. Viene eseguita la runnable **`playWordleSERVER.`**
- *tentativo*: inviato dal client, insieme alla Guessed Word, per tentare di indovinare la Secret Word, all'interno di una partita. Viene eseguita la runnable **`tentativoSERVER.`**
- *exit*: ricevo questo comando sia che l'utente abbia perso la partita finendo i tentativi, sia che l'utente abbandoni la partita quindi la consideriamo come persa. Alla sua recezione il server va ad azzerare lo streak e ad aggiornare la percentuale delle partite vinte di quell'utente, e setta a 1 il flag sulla partita, per indicare che è stata usata per aggiornare le statistiche dell'utente.
- *statistics*: il client lo invia quando vuole ricevere le sue statistiche. Viene eseguita la runnable **`statisticsSERVER`**
- *classifica*: il client lo invia quando vuole ricevere l'attuale classifica. Viene eseguita la runnable **`ShowMeRankingSERVER`**
- *share*: il client invia questo comando dopo che ha finito una partita, per esprimere la sua volontà di condividerla sul gruppo multicast. Viene eseguita la runnable **`ShareServer`**
- *DISCONNETTI*: il client lo invia quando un utente prova a loggarsi ma è già loggato e connesso al server da un altro client, e quindi vuole disconnettersi. Con questo comando si effettua la disconnessione dal vecchio client chiudendo la vecchia SocketChannel e ci si connette al nuovo client. Viene eseguita la runnable **`disconnetti`**
- *connesso*: comando che viene inviato quando un utente effettua il login. Viene eseguita la runnable **`connesso`**

analizziamo adesso queste runnable:

`logout`

Col metodo costruttore vengono presi in input l'oggetto remoto che implementa il servizio di registrazione e di login, il nome utente, estratto dal messaggio di richiesta ricevuto dal client che vuole effettuare il login, e la Selectionkey dal quale verrà estratto il canale da cui è provenuta la richiesta. Per effettuare il logout dell'utente si esegue il metodo **`logout`** sul registro col quale si imposterà la variabile `logged` dell'utente a false. Questa operazione è sincronizzata perché sfrutta il metodo synchronized **`logout`** della classe **`Utente`**. A questo punto, all'interno di un blocco synchronized sulla variabile `connessi`, viene fatta una remove() dell'utente in modo da rimuovere quell'utente da quelli connessi. Si risponde al client, scrivendo sul canale, il messaggio di "ok", si chiude il SocketChannel per la connessione con questo client lato server e si cancella la Selectionkey dal Selection key set del selettore.

`playWordleSERVER`

Il primo controllo che viene fatto è quello di verificare se, nella struttura dati `partite`, l'ultima partita di questo utente è stata usata per aggiornare le statistiche dell'utente, andando a controllare il `flag` della partita. Potremmo trovarci in questa situazione, flag a 1, partita non scaricata, quando un client si disconnette in maniera anomala mentre sta giocando una partita (es Cntrl-c) e quindi il server non riceve nessun comando che gli

permette di usare quella partita per aggiornare le statistiche. A questo punto, all'interno di un blocco synchronized sulla variabile `SWO`, per evitare race condition, controllo se quell'utente ha già giocato la SecretWord del giorno, in caso negativo, lo aggiungo a gli utenti che l'hanno giocata, estraggo la SecretWord, la sua traduzione e il n° di SecretWord, per poi andare a incrementare le partite giocate da quell'utente e scrivere sul canale la risposta "ok:puoi giocare:"+secretword+":"+traduzione". Nel caso in cui risulti che l'utente abbia già giocato la SecretWord il server risponde con la stringa "errore:hai gia giocato la parola del giorno". All'interno di questa Runnable accedo alle strutture dati `pertite` e `registro`, che sono strutture condivise ma sincronizzate perché sono delle ConcurrentHashMap. Inoltre, le modifiche su una partita, di un utente, in `partite` vengono fatte solo da un thread alla volta perché un utente può essere loggato solo su un client e quindi giocare solo da un client, che lavora in maniera sequenziale.

tentativoSERVER

Insieme al comando che avvia questa runnable viene inviata la GuessedWord con la quale si cerca di indovinare la SecretWord. Si controlla che la GuessedWord ricevuta sia contenuta nel vocabolario di Wordle, in caso negativo il server scrive come messaggio di risposta nel canale tentativi_rimasti+":"+"errore2", senza andare a diminuire i tentativi dell'utente. Se invece la parola è contenuta nel vocabolario si aumentano i tentativi fatti nell'oggetto che rappresenta la partita in corso e si diminuiscono i tentativi rimasti. Adesso si controlla se la Guessed Word è uguale alla SecretWord, se lo è, aggiorno le statistiche dell'utente e nell'oggetto `partita` aggiungo il suggerimento "++++++++++" che mi indica che la parola è stata indovinata, setto il `flag` di `partita` a 1 per dire che ho già usato il risultato di questa partita per aggiornare le statistiche. Infine, visto che l'utente ha vinto e ha cambiato così il suo punteggio, devo aggiornare la classifica usando il metodo **`upgrade_classifica`**, che eventualmente andrà anche a inviare una notifica di aggiornamento della classifica. Adesso posso rispondere al client con il messaggio "tentativi_rimasti+":vinto". Se invece la GuessedWord non coincide con la SecretWord si va a creare la stringa suggerimento, la si aggiunge alla lista di suggerimenti della partita e si risponde al client con il messaggio "tentativi_rimasti+":"+"suggerimento:"+suggerimento".

statisticsSERVER

Anche in questo caso controllo che l'ultima partita dell'utente sia stata utilizzata per aggiornare le statistiche dell'utente, prima di inviargliele come stringa attraverso il canale

ShowMeRankingSERVER

Attraverso il metodo **`get_classifica_as_string`** sulla struttura dati `classifica`, ottengo l'intera classifica come stringa, che vado a scrivere sul canale come risposta al client.

ShareServer

Apro all'interno di un try-with-resources un DatagramSocket perché voglio condividere la partita inviando un pacchetto UDP sul gruppo multicast. Ottengo la partita come stringa chiamando il metodo **`get_partita_as_string()`**, metto questa stringa in un array di byte che utilizzerò per creare un DatagramPacket, destinato all'indirizzo e porta del gruppo di multicast. Dopo che ho inviato il pacchetto sulla DatagramSocket, rispondo al client scrivendo sul canale il messaggio "ok".

disconnetti

Per prima cosa si accede a un blocco synchronized sulla variabile `connessi`, per evitare race condition, all'interno del quale, con il metodo get() su `connessi` si recupera la vecchia SocketChannel associata alla connessione con quell'utente. In un if controllo se la get() ha effettivamente restituito qualcosa, potrebbe aver restituito null se non ho nessuna SocketChannel associata a quel nome utente perché nel mentre si è eseguito il logout dal vecchio client. In caso invece la get() abbia restituito un SocketChannel, lo si va a chiudere e da `connessi` con una operazione di put() rimuovo il vecchio SocketChannel e aggiungo quello nuovo relativo alla connessione col nuovo client da cui mi sono connesso. Adesso il server può rispondere al client scrivendo nel canale il messaggio "ok:disconnessione e riconnessione andata a buon fine ".

connesso

In questa runnable si va ad aggiungere il nomeutente, preso in input, alla struttura dati `connessi`, per memorizzarsi quale SoketChannnel è associata alla comunicazione con questo utente. Si risponde al client con la stringa "ok:connessione andata a buon fine" che una volta ricevuta dal client permette il proseguimento nella sua esecuzione sequenziale.