Relazione del progetto di fine corso del corso di "Laboratorio di reti" .
Andrea Lepori, Matricola: 602620, A.A 2022/23 Appello 05/7/2023
Wordle : un gioco di parole 3.0

Breve illustrazione del materiale consegnato e degli strumenti di sviluppo utilizzati.

Il progetto è stato sviluppato ed eseguito con l'ausilio dell'IDE Eclipse, successivamente è stato testato con compilazione ed esecuzione (tramite i comandi javac e java) da terminale Windows 10 (java 17.0.7), su Windows Subsystem for LINUX (WSL) equipaggiato con Ubuntu 22.04.1 (openjdk versione 17.0.7) e su macOS (openjdk 17.0.7.)
La cartella consegnata, denominata progettoLepori, contiene le seguenti sottocartelle :

- client : contenente i codici sorgente (file .java) esclusivi del client, WordleClientMain e WordleClient.
- server : contenente i codici sorgente (file .java) esclusivi del server, nonché WordleServerMain e WordleServer.
- shared : contenente i codici sorgente (file .java) delle classi che sono utilizzate sia dal client che dal server di Wordle.
- libs : contenente i file .jar delle librerie esterne utilizzate nel progetto (gson-2.10.1 e java-json).
- bin : contenente i gli eseguibili (file .class) risultato del processo di compilazione tramite il comando javac del codice sorgente di client e server.

Nella cartella progettoLepori sono presenti anche i seguenti file :

- client_config.properties : file contenente i parametri di configurazione del client.
- server_config.properties : file contenente i parametri di configurazione del server.
- LogRegistrazioni.txt : file contenente testo formato json creato dal thread lato server che effettua la serializzazione degli utenti registrati a Wordle necessario a realizzare la persistenza dello stato del server ed anche a ripristinarne lo stato al riavvio .
- words.txt : file contenente le parole lecite del dizionario di Wordle, una per riga.

Istruzioni per compilare ed eseguire il progetto su vari sistemi operativi.

Windows 10

Per compilare il progetto su Windows 10 seguire le seguenti istruzioni : aprire un terminale nella cartella progettoLepori oppure aprire il terminale in una cartella qualsiasi e posizionarsi nella cartella progettoLepori. Dalla cartella progettoLepori compilare server e client digitando :

```
javac -cp ".\libs\*" -d .\bin\ -sourcepath ".\server\;.\shared"
.\server\WordleServerMain.java

javac -cp ".\libs\*" -d .\bin\ -sourcepath ".\client\;.\shared" .\client\WordleClientMain.java
```

Dalla cartella progettoLepori eseguire prima(*) il server digitando :

```
java -cp ".\libs\*;.\bin\" WordleServerMain
```

Per eseguire il client aprire un altro terminale e sempre posizionandosi nella cartella progettoLepori digitare :

java -cp ".\libs*;.\bin\" WordleClientMain

(*)per il corretto funzionamento del progetto è necessario lanciare prima il server e poi il client.

Ubuntu e MacOS

Per compilare il progetto su Linux Ubuntu o MacOS seguire le seguenti istruzioni: aprire il terminale nella cartella progettoLepori oppure aprire il terminale in una cartella qualsiasi e posizionarsi nella cartella progettoLepori. Dalla cartella progettoLepori compilare server e client digitando :

javac -cp ./libs/gson-2.10.1.jar:./libs/java-json.jar -d ./bin/ -sourcepath ./server /:./shared/ ./server/WordleServerMain.java

javac -cp ./libs/gson-2.10.1.jar:./libs/java-json.jar -d ./bin/ -sourcepath ./client/:./shared/ ./client/WordleClientMain.java

Dalla cartella progettoLepori eseguire prima(*) il server digitando :

java -cp ./libs/gson-2.10.1.jar:./libs/java-json.jar:./bin/ WordleServerMain

Per eseguire il client aprire un altro terminale e sempre posizionandosi nella cartella progettoLepori digitare :

java -cp ./libs/gson-2.10.1.jar:./libs/java-json.jar:./bin/ WordleClientMain

(*)per il corretto funzionamento del progetto è necessario lanciare prima il server e poi il client.
Come richiesto, sono stati creati due file con l'estensione jar, uno per il server (ServerWordle.jar) ed uno per il client(ClientWordle.jar). Questi file sono stati creati su WSL equipaggiata con Ubuntu ed eseguiti sulla stessa. Di seguito i comandi utilizzati per creare i file .jar e per eseguirli.

NOTA : i seguenti comandi sono stati eseguiti a partire dalla directory centrale del progetto progettoLepori. Prima di eseguire i seguenti comandi è necessario creare i due file manifest per il server e per il cliet (MANIFESTSERVER.TXT e MANIFESTCLIENT.TXT) nella directory centrale del progetto

Per creare il jar del server :

mkdir buildServer

find ./server/ shared/ -name "*.java" > sourcesServer.txt

javac -cp ./libs/java-json.jar:./libs/gson-2.10.1.jar -d buildServer/ @sourcesServer.txt

cd buildServer/

jar mcfe ../MANIFESTSERVER.txt ../ServerWordle.jar WordleServerMain *

cd ..

Per eseguire il jar del server

java -cp ./ServerWordle.jar:./libs/gson-2.10.1.jar:./libs/java-json.jar:. WordleServerMain

Per creare il jar del client :

mkdir buildClient

find ./client/ shared/ -name "*.java" > sourcesClient.txt

javac -cp ./libs/java-json.jar:./libs/gson-2.10.1.jar -d buildClient/ @sourcesClient.txt

cd buildClient

jar mcfe ../MANIFESTCLIENT.txt ../ClientWordle.jar WordleClientMain *

cd ..

Per eseguire il jar del client :

java -cp ./ClientWordle.jar:./libs/gson-2.10.1.jar:./libs/java-json.jar:. WordleClientMain

Guida all' uso dei comandi dell'interfaccia testuale del client

```
**********************************************
Wordle: un gioco di parole 3.0
Se hai già un account fai login per giocare
Se non sei ancora registrato effettua la registrazione
**********************************************
Per effettuare il login digita: login
Per effettuare la registrazione digita: registrazione
```

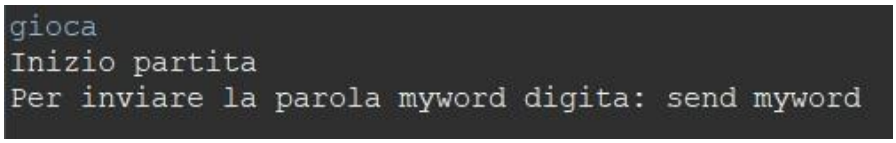
Il client di Wordle mette a disposizione un' interfaccia testuale da linea di comando per interagire col gioco. Una volta avviato il client nella schermata comparirà un messaggio in cui sarà richiesto al giocatore di procedere scegliendo una tra due opzioni disponibili: se il giocatore è già registrato a Wordle potrà accedere al gioco digitado il comando **login** e schiacciando invio. Successivamente verrà poi richiesto all' utente di inserire prima il proprio username e poi la propria password. Se il giocatore deve ancora registrarsi, dovrà farlo prima di eseguire il login, per registrarsi è necessario digitare il comando **registrazione** e premere invio. Per completare la registrazione verrà sempre richiesto di inserire prima un username e poi una password. Di seguito un esempio di interazione :

```
Per effettuare il login digita: login
Per effettuare la registrazione digita: registrazione
registrazione
Inserisci il tuo username
mario
Inserisci la tua password
rossi
Il tuo account è stato creato, per accedere digita login
login
Inserisci il tuo username
mario
Inserisci la tua password
rossi
Login eseguito con successo

Registering for callback
Unito al gruppo di multicast
Benvenuto su Wordle mario!
Per giocare digita: gioca
Per effettuare il logout digita: logout
Per visualizzare le tue statistiche  digita: statistiche
Per visualizzare le classifica di Wordle digita: classifica
Per richiedere la condivisione dei suggerimenti sul gruppo di Multicast digita : share
Per visualizzare i suggerimenti ricevuti sul gruppo di Multicast digita : ShowMeSharing
Per uscire digita: exit
connected to server
```

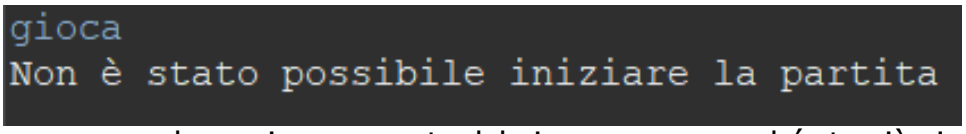
Una volta che un utente si è loggato con successo, o passando direttamente dalla login oppure prima registrandosi e poi eseguendo la login, compariranno due messaggi "Registering for callback" e "Unito al gruppo di multicast" che comunicheranno rispettivamente all' utente l'avvenuta registrazione al servizio di callback per le notifiche di variazione del podio e l'iscrizione al gruppo di multicast su cui sono inviate le condivisioni

dei suggerimenti dai vari utenti del gioco. Contestualmente viene mostrato un messaggio di benvenuto e il menù testuale con tutte le istruzioni per interagire col client.

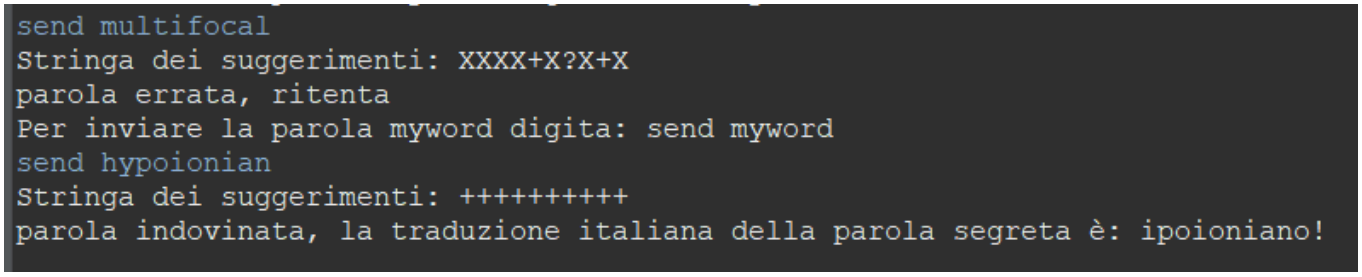


Attraverso il comando **gioca** sarà possibile richiedere l'avvio di una partita per la sessione corrente del gioco.

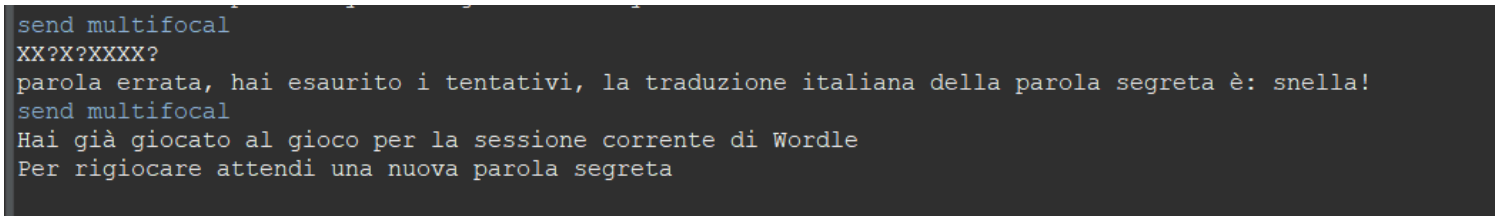
Se è stato possibile avviare un partita, verrà stampato il messaggio "Inizio partita" e un messaggio con le istruzioni per inviare una parola (guessed word), per cercare di indovinare la parola segreta (secret word). Per inviare una parola, è necessario digitare il comando **send** e separata da uno spazio la parola che si vuole inviare, ad esempio per inviare la parola multifocal bisogna digitare send multifocal e schiacciare invio.



Se non è stato possibile iniziare una nuova partita perché l'utente ha già giocato per la sessione corrente del gioco oppure perché sta già giocando, verrà mostrato il messaggio nell'immagine e si dovrà attendere l'inizio di una nuova sessione con una nuova parola segreta per richiedere di giocare con successo.



Dopo aver inviato la parola multifocal col comando send multifocal, in base all'esito del tentatitivo, vitttorioso o perdente, l' interfaccia ci risponderà in modo diverso mostrando, tra le altre cose, insieme all'esito del tentativo, la stringa dei suggerimenti rispetto alla guessed word inviata e alla secret word corrente e in caso di vittoria ci verrà mostrata anche la traduzione italiana della parola segreta.



Una volta effettuato il 12-esimo tentativo, abbiamo terminato i tentativi a nostra disposizione per indovinare la parola nella sessione corrente, perciò la partita è considerata terminata e persa ma viene lo stesso fornita la traduzione della parola segreta. Dopo che una partita è terminata, vinta o persa che sia, tutti i tentativi di invio di una parola non saranno considerati validi e sarà stampato a video un messaggio che spiega che abbiamo già giocato per quella sessione e di aspettare la successiva per rigiocare.

```
send parolanonindizionario
Hai inviato una parola che non fa parte del vocabolario di Wordle
Inviane un' altra, ricordati che sono ammesse solo parole di 10 caratteri
```

Se viene inviata una parola non appartenente al dizionario o di lunghezza sbagliata, il tentativo non inficia sul totale dei tentativi a disposizione e viene richiesto l'invio di un'altra parola.

Per eseguire il logout bisogna digitare il comando **logout** .

```
logout
Logout eseguito con successo
```

Per richiedere la stampa delle statistiche dell'utente nel gioco bisogna digitare il comando **statistiche**

Per visualizzare la classifica del gioco bisogna digitare il comando **classifica**. La classifica mostra ordinati dal primo all' ultimo gli utenti del gioco assieme alle loro statistiche, la password è coperta da una serie di '*'.

```
statistiche
partite : 5
win_rate : 80.0
current winning streak : 0
max winning streak : 4
+distribution of attempt to win the game :
+1 attempt : 50.0
+2 attempt : 0.0
+3 attempt : 25.0
+4 attempt : 0.0
+5 attempt : 0.0
+6 attempt : 25.0
+7 attempt : 0.0
+8 attempt : 0.0
+9 attempt : 0.0
+10 attempt : 0.0
+11 attempt : 0.0
+12 attempt : 0.0
+
```

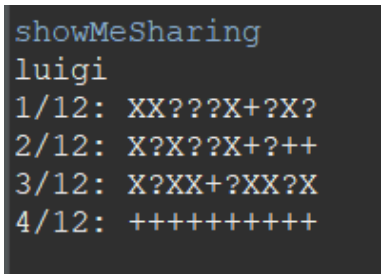
```
classifica
1°: username :a
password :*
logged :false
score :3.0
statistiche :partite : 11
win_rate : 54.545456
current winning streak : 0
max winning streak : 5
+distribution of attempt to win the game :
+1 attempt : 33.333336
+2 attempt : 50.0
+3 attempt : 0.0
+4 attempt : 16.666668
+5 attempt : 0.0
+6 attempt : 0.0
+7 attempt : 0.0
+8 attempt : 0.0
+9 attempt : 0.0
+10 attempt : 0.0
+11 attempt : 0.0
+12 attempt : 0.0
+

2°: username :pinco
password :*****
logged :true
score :3.0
statistiche :partite : 11
win_rate : 54.545456
current winning streak : 1
max winning streak : 2
+distribution of attempt to win the game :
+1 attempt : 33.333336
+2 attempt : 33.333336
+3 attempt : 33.333336
+4 attempt : 0.0
+5 attempt : 0.0
+6 attempt : 0.0
+7 attempt : 0.0
+8 attempt : 0.0
+9 attempt : 0.0
```

Per richiedere la condivisione dei suggerimenti, relativi alle guessed word inviate nel corso di una partita di Wordle sul gruppo di multicast, digitare il comando: **share**. Attenzione è possibile solamente condividere i suggerimenti relativi ad una partita terminata nella sessione corrente soltanto prima del termine della sessione stessa, quindi possiamo condividere i suggerimenti relativi ai nostri invii di guessed word digitando il comando share

solo dopo aver terminato la partita e prima che la sessione cambi, la richiesta di condivisione dei suggerimenti di una partita giocata in una sessione precedente a quella corrente non avrà effetti.

Per visualizzare i suggerimenti inviati dagli utenti sul gruppo di multicast, relativi alle partite di cui hanno condiviso i risultati mediante il comando share,digitare il comando **showMeSharing**, se sono arrivati suggerimenti sul gruppo di multicast questi verranno mostrati a schermo.



Per terminare il client digitare il comando **exit**

Breve descrizione del progetto e panoramica sull' architettura generale dei programmi e sulle scelte implementative effettuate.

Il progetto del gioco di Wordle che ho realizzato consta essenzialmente di due unità logiche : un client e un server.
Il compito del client è quello di interfecciarsi con l'utente del gioco, recepire le sue richieste, prenderle in carico e gestirle comunicando col server del gioco, su cui risiede il cuore, la logica del gioco di Wordle e, una volta ricevute le risposte, informare l'utente dell' esito dei comandi impartiti e guidare quelli che possono essere i successivi passi nel gioco.
Il client è perciò un "thin client", implementa un set minimale di funzioni, l'interfaccia testuale con l'utente e la comunicazione col server del gioco.
Il server di Wordle implementa la logica del gioco, ma non solo! Il server, appoggiandosi ad opportune classi, gestisce il database delle registrazioni, la classifica del gioco, il servizio di notifica di variazione del podio, l' invio delle condivisioni dei suggerimenti delle partite su un gruppo di multicast che conferisce un aspetto social al gioco, oltre a curare la gestione vera e propria del gioco di Wordle con tutta le strutture per la gestione delle varie sessioni di gioco, i metodi per il loro aggiornamento e la gestione della comunicazione coi client.
Il client di Wordle viene avviato nella classe WordleClientMain, ed è sempre qui che sono recuperati i parametri di settaggio da un apposito file di configurazione. Una volta avviato, il client forza l' utente a loggarsi, o direttamente, o previa registrazione se non ha già un account, registrazione e login sono implementati tramite il meccanismo di java RMI, invocando da remoto i metodi dell' oggetto che implementa il servizio di registrazione che risiede sul server. In seguito alla login, il client si iscrive al meccanismo di callback RMI sempre attraverso l'invocazione remota di metodi del servizio di gestione delle notifiche che risiede sul server e viene avviato il thread incaricato di iscriversi e di stare in ascolto sul gruppo di multicast. Sempre dopo la login viene instaurata la connessione tcp col server, sulla quale viaggiano le richieste/risposte che implementano il protocollo di

comunicazione stabilito. In base al comando impartito al client dall' utente sarà invocato un opportuno metodo, per ogni possibile comando è stato scritto un metodo atto a gestirlo. La struttura dei metodi che gestiscono i vari comandi è pressochè la stessa per tutti i comandi e prevede l'invio di un messaggio al server secondo il protocollo stabilito sul socket della connessione TCP sfruttando il Java I/O , quindi gli stream (InputStream e OutputStream, magari avvolti in qualche wrapper come BufferedReader, DataOutputStream, DataInputStream per facilitare la programmazione), l' attesa della risposta del server e la stampa a schermo delle informazioni ricevute. Lato client perciò viene usato un meccanismo di I/O bloccante come gli stream di Java I/O, questo perché perché essendo il client minimale, non deve fare altro che scambiarsi messaggi col server, senza fare nulla in più, non può proseguire nel suo lavoro finchè non ha inviato una richiesta e ricevuto la risposta, quindi tanto vale bloccarsi in attesa.
Il server invece ha una struttura più articolata perché, in linea teorica, deve poter gestire tante connessioni verso tanti client, quindi è preferibile che riesca a lavorare con un alto grado di parallelismo, gestendo contemporaneamente richieste di natura diversa provenienti da client diversi.
Per fare questo il server, avviato sempre dal WordleServerMain che acquisisce i parametri di setting dal file di configurazione del server, si basa su un threadpool con una dimensione fissa del numero di thread, uguale al numero di core della macchina, ai quali saranno passati, per essere eseguiti, vari task che andranno a gestire tutte le possibili richieste in arrivo dal client. La scelta della tipologia del threadpool è ricaduta sul FixedThreadpool perché si vuole garantire un giusto trade-off tra il parallelismo introdotto dal multithreading e il costo in risorse introdotto dall' alto numero di thread e il decadimento delle prestazioni dovuto al context-switching. Gli svantaggi di questa soluzione si possono palesare nel momento in cui si presenta un picco di utenti connessi al gioco, in questa situazione il FixedThreadpool potrebbe andare in crisi, si potrebbe pensare a quel punto di utilizzare un threadpool il cui numero di thread cresca dinamicamente col crescere del numero delle richieste di task da eseguire. Come abbiamo detto, il server mantiene al suo interno gli oggetti che gestiscono il servizio di registrazione, di notifica, la classifica e il gioco di Wordle. Al momento dell' avvio il server crea e starta due thread che periodicamente, con una cadenza stabilita da parametri di configurazione, andranno uno a realizzare la persistenza dello stato, quindi la serializzazione in oggetti json delle strutture che servono per permettere il riavvio del server, mantenendo traccia di quelli che erano gli utenti registrati e la loro classifica e un altro che implementa il refresh del gioco, quindi la chiusura della sessione in corso, con le partite non ancora finite che devono essere terminate e considerate perse, il reset delle strutture che caratterizzano la sessione, ovvero gli utenti in gioco in quella sessione con le loro partite e l'avvio di una nuova sessione con una nuova parola segreta e una nuova traduzione. Lato server, la comunicazione col client è implementata col meccanismo di Java New I/O (NIO), quindi parliamo di channel, col selector per il multiplexing dei canali, meccanismo che mi permette di accettare le richieste di connessione provenienti dai client e di leggere le loro richieste in maniera non-bloccante. Ciò è di vitale importanza, se lato server avessi usato un meccanismo di I/O bloccante per accettare le connessioni e leggere le richieste, le prestazioni del server sarabbero state condizionate da un collo di bottiglia individuabile nella connessione più lenta, mi sarebbe bastata una singola connessione molto lenta per portare al decadimento delle prestazioni. Dopo aver letto la richiesta e parsato il comando inviato dal client, consegno al mio threadpool il task corrispondente da eseguire. L' esecuzione dei task sarà quindi parallela proprio per la definizione di threadpool. Sarà un thread del mio pool ad eseguire il task inviato dal client, eseguendo le

opportune operazioni sulle strutture dati condivise (questo introduce dei problemi di sincronizzazione su queste strutture, dovuti all' accesso concorrente da thread multipli) e una volta gestito ciò che doveva essere fatto il task dovrà mandare un messaggio di risposta al client sul socketChannel che identifica la connessione con quel particolare client, per comunicare l'esito delle operazioni ed eventuali altre informazioni. Questa è per sommi capi l'architettura dei programmi client e server.

Breve descrizione delle classi del progetto nonché delle strutture dati utilizzate lato server e lato client.

WordleClientMain

La classe WordleClientMain contiene il metodo main attraverso il quale viene mandato in esecuzione il client di Wordle. Il metodo main di questa classe si incarica anche di andare a leggere i parametri di configurazione del client dal relativo file di configurazione. Nel metodo main viene istanziato un oggetto WordleClient client che implementa il client, al cui costruttore vengono passati i parametri di configurazione precedentemente letti. Il client viene successivamente avviato eseguendo il metodo client.start().

WordleClient

La classe WordleClient fornisce l'implementazione del client di Wordle. La classe contiene il metodo start() che viene invocato per avviare il client. Per prima cosa prendo il registry dove reperire gli stub degli oggetti remoti che mi serviranno. Una volta preso il registro recupero lo stub del servizio di registrazione (RegistrationService) facendo una lookup nel registry. Quindi entro in un ciclo while, con guardia (!logged), difatti finchè non sono loggato non posso giocare, attraverso il metodo next() dello scanner prendo in input il comando inserito da tastiera dall' utente che può essere "login" per loggarsi o "registrazione" per registrarsi. In un primo switch-case gestisco il login e la registrazione, per il login prendo username e password forniti da tastiera che passo all'opportuno metodo login, insieme allo stub del servizio di registrazione, questo metodo gestisce il login attraverso l'uso di RMI. Se il login va a buon fine, reperisco lo stub del servizio di notifica (NotificationService) dal registry e iscrivo il client al servizio di callback. Inoltre creo e avvio il thread gestoreMulticast che esegue il task MulticastManager. Il task MulticastManager implementa il task che iscrive un client al gruppo di multicast e sta in ascolto sul gruppo di multicast per ricevere i messaggi. Nell' overrirde del metodo run, dormo per 1 minuto, mi sveglio e leggo un datagramma dal multicastSocket, il cui messaggio con i suggerimenti viene convertito in una stringa che aggiungo alla lista che memorizza lato client i messaggi provenienti dal gruppo di Multicast. La registrazione viene gestita con il metodo register che implementa la registrazione attraverso il meccanismo di Java RMI, effettuando chiamate di metodo remoto all' oggetto remoto che implementa il servizio di registrazione. Una volta uscito dal primo ciclo while un utente è correttamente

loggato, possiede un username e una password. Per prima cosa creo una connessione TCP con il server, quindi creo un socket e lo connetto su localhost, alla porta specificata da file di configurazione e realizzo così la connessione TCP col server. Su questo socket apro un InputStream e un OutputStream, rispettivamente per leggere e scrivere byte su questa connessione. Per semplificare la successiva lettura e scrittura su questi due stream li "avvolgo" rispettivamento in un DataInputStream e un DataOutputStream , inoltre creo un BufferedReader su un InputStreamReader creato sull' InputStream. A questo punto si arriva al cuore del client , finchè non diventa vera una variabile booleana stop leggo il comando inserito da tastiera dall'utente e lo gestisco attraverso uno switch-case. Il comando inserito dall' utente può essere "gioca" per richiedere l'avvio di una partita di Wordle, "logout" per eseguire il logout, "exit" per terminare il client dopo essere uscito dal ciclo ponendo stop a true, "send" per spedire una guessed word anch' essa inserita da tastiera, "statistiche" per visualizzare le statistiche di gioco, "classifica" per visualizzare la classifica del gioca, "share" per richiedere la condivisione dei suggerimenti relativi alla propria partita sul gruppo multicast, e "showMeSharing" per visualizzare i suggerimenti inviati dai vari utenti sul gruppo multicast. La classe WordleClient mette a disposizione vari metodi che servono a gestire le varie richieste che un utente può rivolgere al client, di seguito sono elencati i principali ;

- il metodo showMesharing serve a visualizzare le condivisioni inviate dagli utenti sul gruppo di multicast e memorizzate nella List<String> multicastMessage, se nella lista sono presenti elementi questi vengono stampati e rimossi uno ad uno, altrimenti viene stampato a terminale un messaggio che indica che non sono disponibili condivisioni;
- il metodo share serve per richiedere la condivisione dei suggerimenti riferiti alla propria partita, appena terminata, sul gruppo di multicast. Questa richiesta verrà soddisfatta se e solo se la richiesta di share avverrà dopo aver terminato con vittoria o sconfitta una partita e prima che inizi la sessione successiva del gioco. Nel corpo del metodo preparo il messaggio da spedire secondo il protocollo da me stabilito (username/password/share), sempre come da protocollo invio sull' outputstream associato al socket TCP l'intero che indica il numero dei byte del messaggio che il server dovrà leggere, invio il messaggio e poi faccio un flush(). A questo punto attendo di leggere la risposta dal server, facendo una reader.readLine(), da protocollo il server invia linee in risposta, ovvero stringhe terminate da '/n'. Parso la risposta del server che, se tutto va bene, mi risponde con username/share/ok mentre se l'invio sul gruppo multicast non è stato fatto risponde con username/share/no;
- il metodo logout implementa la richiesta di logout dell' utente con le credenziali (username e password) passate in input, comunica al server, secondo protocollo sulla connessione TCP, di eseguire il logout per quell' utente e attende la risposta del server, se la risposta è positiva il logout è stato effettuato e viene restituito true, false altrimenti.

- Il metodo login, come abbiamo già visto esegue il login dell' utente con username e password passati in input attraverso la chiamata dei metodi remoti del servizio di registrazione acceduto tramite il suo stub, passato anch' esso come parametro al metodo, secondo il meccanismo di Java RMI. Il metodo restituisce true se e solo se il login è andato a buon fine, false se si sono verificati delle condizioni per cui non è stato possibile eseguirlo (utente non registrato, password sbagliata);
- Il metodo register prende in input le credenziali dell'utente che vuole registrarsi e lo stub del RegistrationService e invoca attraverso di esso il metodo AddRegistration dell' oggetto remoto, restituisce true se e solo se la registrazione è andata a buon fine.
- Il metodo playWordle permette al client di comunicare al server che l'utente individuato da username e password vuole iniziare una partita a Wordle. Questo metodo restituisce true se e solo se è stato possibile avviare una nuova partita, false altrimenti.
- Il metodo sendWord gestisce l'invio di una guessed word al server.Invio il messaggio di richiesta sempre seguendo il mio protocollo di comunicazione (da notare che nel messaggio invio anche la guessed word) e attendo la risposta nel solito modo, leggendo una linea. Se la risposta è affermativa è stata indovinata la parola segreta della sessione corrente di Wordle, quindi il metodo stampa a schermo un messaggio in cui si dice che si è indovinato la parola e se ne mostra la traduzione italiana, inviata dal server nella risposta. Se nel codice della risposta del server è presente la stringa "ritenta", la parola non si è indovinata e si indica che si può effettuare un altro tentativo, se invece il codice è "rinvia" vuol dire che era stata inviata una parola non presente nel dizionario o della lunghezza sbagliata per questo viene stampato a schermo di rinviare un' altra parola. Il codice "fine" mi dice che i tentativi consentiti sono esauriti ed infine il codice "gioca" mi dice che prima di inviare una parola devo avviare una partita.
- showMeRanking : metodo con cui un client richiede la visualizzazione a schermo della classifica del gioco. Da notare il meccanismo diverso usato per ricevere la risposta dal server. Nell' intero bytes_to_read viene letto l' intero che mi dice quanti bytes andranno letti nel successivo messaggio. Poi alloco char[] array che è un array di caratteri con tanti elementi quanti sono i bytes_to_read. Inizializzo a 0 una variabile intera bytes_read che fungerà da contatore dei bytes letti. A questo punto in un while, leggo da un BufferedReader nell' array di char finchè non ho letto tutti i bytes del messaggio, converto l'array di caratteri in una stringa che conterrà la classifica che poi stampo.
- sendMeStatistics : metodo attraverso cui il client chiede di visualizzare a schermo le proprie statistiche nel gioco.

NotifyEventInterface (interfaccia)

NotifyEventInterface è un' interfaccia che contiene la dichiarazione di un metodo notifyEvent invocabile da remoto che serve per notificare il verificarsi di un particolare evento al client.

NotifyEventImpl

Classe che implementa l'interfaccia NotifyEventInterface perciò fornisce la definizione del metodo notifyEvent che può essere richiamato da remoto dal servente per notificare il verificarsi di un determinato evento che in questo è la variazione del podio. Il metodo stampa a schermo la notifica della variazione del podio, memorizza il nuovo podio nella struttura che lo memorizza lato client e lo stampa a schermo.

Utente

La classe Utente rappresenta un utente registrato. Un utente è individuato univocamente da una stringa **username** e ha una **password** che è rappresentata da una stringa non vuota. Un utente registrato a Wordle può essere, in un certo istante di tempo, loggato o non loggato, questa informazione è registrata dal flag boolean **logged** che vale true se e solo se l'utente è attualmente loggato. La variabile **score** (float) mantiene il punteggio dell' utente, mentre le statistiche di interesse sono memorizzate nella variabile **stats** della classe Statistiche. Il metodo costruttore della classe Utente prende in input due stringhe rispettivamente username e password, bisogna controllare prima di passare il parametro che contiene la stringa username al costruttore che questa sia univoca all' interno del database degli utenti, mentre l'inserimento di una stringa vuota per la password provoca il lancio di un' eccezione IllegalArgumentException. La classe dispone di opportuni metodi getter per reperire i valori della variabili private locali e di opportuni metodi setter per modificarne i valori. Dato che un oggetto Utente potrebbe essere una risorsa condivisa tra più thread, i metodi "critici", ovvero quelli che potrebbero dare luogo a race condition dovute all'accesso concorrente, sono protetti grazie all'accesso in lettura e scrittura in mutua esclusione attraverso il costrutto Java synchronized. La classe Utente dispone di un metodo **updateScore** per il ricalcolo e l'aggiornamento del punteggio. Il punteggio viene calcolato moltiplicando il numero di vittorie dell' utente per un fattore di penalità. Il fattore di penalità è pari a 1/(numero medio dei tentativi nelle partite vincenti) ed è tale da avvantaggiare nel punteggio gli utenti che impiegano un basso numero di tentativi per indovinare la parola segreta. La classe Utente implementa l'interfaccia Comparable<Utente> e quindi ridefinisce il metodo CompareTo per stabilire una relazione di ordinamento tra gli utenti. Due utenti sono uguali se hanno lo stesso username, mentre l'utente u1 precede nell'ordinamento stabilito l'utente u2 (u1<u2) se lo score di u1 è minore dello score di u2, a parità di score vale l'ordinamento lessicografico degli username. Inoltre la classe Utente fornisce l'override del metodo equals che sostanzialmente richiama il metodo equals, definito per le stringhe, sugli username dei due utenti di cui si vuole verificare l'uguaglianza e ridefinisce anche il metodo toString per

fornire una rappresentazione testuale, sotto forma di stringa delle principali informazioni di interesse dell' utente, la stampa della password viene coperta con una serie di '*'.

Statistiche

La classe Statistiche racchiude le statistiche di interesse per un utente. Le statistiche di interesse sono il numero di **partite** giocate, il numero di **vittorie** ottenute, il **win rate** (percentuale di vittorie sul totale delle partite giocate), la **current win streak** e la **max win streak** rispettivamente l' attuale e la massima striscia di vittorie consecutive, la **guess distribution** è la distribuzione dei tentativi impiegati per indovinare una parola segreta sul totale delle parole segrete indovinate ed è definita per gli interi da 1 a 12 compresi, che sono rispettivamente il minimo ed il massimo numero di tentativi impiegabili per arrivare alla soluzione del gioco. Sono definite inoltre alcune variabili di supporto che risultano utili nelle varie funzioni di aggiornamento delle statistiche e del punteggio come l'intero attempts che memorizza la somma complessiva dei tentativi impiegati per arrivare alle varie soluzioni nelle partite vincenti e l'array di interi attemps_to_win che memorizza sempre nella posizione "i-esima" dell'array, per "*i*" che va da 0 a 11 ,non la distribuzione, ma il numero complessivo di partite vinte con "*i*+1" tentativi. Nel metodo costruttore della classe tutte le variabili vengono inizializzate a 0. La classe mette a disposizione gli opportuni metodi getter e setter per accedere e modificiare i valore delle variabili private locali. Dato che gli oggetti della classe statistiche potrebbero essere risorse condivise fra vari thread, i metodi getter e setter sono protetti da possibili race condition dovute all'accesso concorrente attraverso l'utilizzo del modificatore synchronized di Java. Il metodo update da invocare una volta terminata una partita per l'aggiornamento delle statistiche di un utente, agisce in funzione di un parametro booleano win che indica se la partita appena terminata è stata vinta o persa e in base a questa informazione aggiorna le statistiche nel modo opportuno. Sia che abbia vinto sia che abbia perso la partita, il metodo incrementa il numero di partite giocate, se ho vinto incremento la somma complessiva dei tentativi impiegati nelle partite vittoriose del valore intero tentativi passato come parametro al metodo, incremento di uno il numero di vittorie e la striscia corrente di vittorie consecutive, registro il fatto che la partita è stata vinta con tot tentativi incrementando di uno il valore presente alla posizione (tentativi-1) nell'array attempts_to_win e aggiorno la guess_distribution. Inoltre se l'attuale striscia di vittorie consecutive è più grande del valore massimo della striscia di vittorie consecutive faccio l'aggiornamento. Se la partita è persa azzero il valore della current_win_streak. La ridefinizione del metodo toString fornisce una rappresentazione delle statistiche sotto forma di stringa.

Game

La classe Game rappresenta il gioco di Wordle. Il gioco sta nel dover indovinare una **secret word** (parola inglese di 10 lettere, presa da un dizionario fornito al gioco), la secret Word cambia ad ogni nuova sessione del gioco. Gli utenti che hanno effettuato o stanno effettuando una partita nella sessione corrente del gioco sono memorizzati nella

struttura **players, Map<Utente, Partita>** , dove l'oggetto Utente agisce da chiave mentre la Partita che sta giocando o ha giocato è il valore associato. Il gioco dispone di un **vocabolario** di parole lecite implementato come una List<String>, il gioco ha come variabile **classifica,** un oggetto della classe Classifica che implementa la classifica del gioco, si mantiene una stringa che è la **traduzione** italiana della parola segreta corrente e un intero **time_to_awayt** che indica il tempo dopo il quale deve essere effettuato il refresh del gioco, espresso in ms. Inoltre, la variabile privata **ns** della classe NotificationServiceImpl mantiene un riferimento all'oggetto che implementa il servizio di notifica che serve ad inviare una notifica a tutti gli utenti loggati ogni qualvolta si verifica una variazione nel podio della classifica del gioco. Il costruttore della classe inizializza il dizionario come una LinkedList mentre la struttura players come una ConcurrentHashMap, poiché la risorsa dovrà essere condivisa tra più thread l' utilizzo di una struttura presincronizzata da Java mi esenta dalla sincronizzazione esplicita. Apro in lettura il file che contiene le parole del dizionario da cui leggo una linea alla volta, prendo la parola presente in quella linea e l'aggiungo alla struttura che implementa il dizionario. Genero una parola segreta e la sua traduzione. L'aggiornamento del gioco, che prevede il cambio della parola segreta quindi la terminazione della vecchia sessione e l'inizializzazione di una nuova, è realizzato mediante un thread che esegue il task refreshGame, creato e startato nel costruttore della classe gioco. Il task refreshGame è implementato come una classe interna che implementa l'interfaccia Runnable, nell' overrirde del metodo run (synchronized) si entra in un ciclo infinito in cui il thread dorme per il tempo necessario (tempo di vita di una sessione con una data parola segreta), si sveglia, termina la sessione corrente terminando forzatamente le partite ancora in corso che vengono considerate perse, resetta la struttura dati players che mantiene gli utenti che stanno partecipando alla sessione corrente del gioco con le rispettive partite, cambia la parola segreta e aggiorna la traduzione, si rimette poi a dormire. Dato che la struttura game può essere condivisa tra più threads i suoi metodi sono dichiarati synchronized. La classe mette a disposizione gli opportuni metodi getter per ottenere la parola segreta corrente del gioco e la sua traduzione, la classifica del gioco, il metodo getPartita che preso un Utente u come parametro restituisce la partita associata ad u nella sessione corrente se questa esiste, null altrimenti, il metodo alreadyPlayed preso in input un Utente u restituisce true se e solo se esiste un mapping per quell' Utente u nella struttura players per l'attuale sessione (ovvero l'utente sta partecipando all' attuale sessione del gioco, attenzione la partita associata può essere in corso o terminata), il metodo inVocabulary presa una stringa restituisce true se quella parola è contenuta nel vocabolario del gioco, false altrimenti. La classe mette a dispozioni opportuni metodi setter per lavorare con le variabili e le strutture private del gioco, addPlayer prende in input un utente e lo aggiunge alla struttura players associandogli un oggetto Partita, se l'utente non sta già giocando al gioco (!this.alreadyPlayed(utente) == TRUE), se questa condizione non vale viene sollevata una IllegalArgumentException, resetPlayer() resetta la struttura players che mantiene il riferimento alla ConcurrentHashMap che memorizza i giocatori di Wordle con le rispettive partite per la sessione corrente istanziandone una nuova, invocando il costruttore, setSecretWord e setTranslation fanno la cosa ovvia. Tra i metodi di supporto troviamo il

metodo suggestion che presa la guessed word restituisce la Stringa dei suggerimenti ottenuta confrontando la gw con la sw corrente, il metodo Translate prende in input una stringa word (parola inglese) e restituisce un'altra stringa che è la traduzione italiana di word ottenuta effettuando una query http al servizio *https://mymemory.translated.net/doc/spec.php* e andando a prendere la traduzione nel campo opportuno della risposta http, generateRandomWord restituisce una parola a caso tra quelle presenti nel dizionario, il metodo test prende in input un Utente u e una String guessedWord, reperisce ,se presente nella struttura players (che mantiene gli utenti con le rispettive partite per la sessione corrente del gioco), la partita associata ad u, innanzitutto registra il tentativo effettuato chiamando il metodo addTentativo sull' oggetto partita, poi se ho indovinato la parola (guessedWord.equals(secretWord)), registro che la partita è terminata e che è stata vinta altrimenti se non ho indovinato controllo se ho raggiunto il limite massimo di tentativi concessi e in tal caso registro la fine della partita e la sconfitta, se non ho indovinato e non ho esaurito i 12 tentativi non faccio niente. Con un ultimo controllo guardo se la partita è terminata (vinta o persa che sia), aggiorno le statistiche chiamando il metodo update a cui passo gli opportuni parametri e dopo aggiorno lo score dell' utente con updateScore. A questo punto dato che il cambiamento dello score potrebbe aver scombinato l'ordine corretto della classifica ordino chiamando il metodo sort che, tra l'altro, mi restituisce un booleano che indica se il podio è variato , in tal caso chiamo il metodo opportuno dell'oggetto NotificationServiceImpl ns che si occuperà di fare le callback per informare della variazione tutti gli utenti loggati. Il metodo endGame termina "abrupt" una sessione di gioco, terminando forzatamente le partite ancora in corso, considerandole perse, lo fa scorrendo tutte le Map.Entry<Utente, Partita> di players.entrySet(), se dato p = entry.getValue() vale che !(p.isEnd()) ovvero la partita non è ancora finita, la termino e la setto come persa, aggiorno prima statistiche e poi score dell' utente e una volta terminato questo ciclo ordino e controllo le variazioni nel podio.

Partita

Questa classe rappresenta una partita di Wordle. Le variabili private locali di questa classe sono **tentativi**, una List<String> che memorizza i tentativi fatti da un utente per indovinare la parola segreta attraverso la stringa dei suggerimenti generata rispetto alla guessed word inviata e alla secret word attuale, un intero **attempts** che conta i tentativi (massimo 12), un booleano **end** che varrà true se e solo se la partita è terminata. E un booleano **win** che varrà true se e solo se la partita è vinta, questo booleano è significativo se e solo se end == true ovvero se la partita è terminata. Il costruttore inizializza tentativi come un ArrayList , i booleani win e end sono settati a false e attempts a 0. La classe mette a disposizoni i metodi getter e setter neccessari per manipolare le variabili locali e le strutture private , inoltre dato che un oggetto partita potrà essere una risorsa condivisa tra più threads tutti i metodi della classe sono dichiarati synchronized.

Classifica

Classe che realizza la classifica di Wordle. La **classifica** è implementata come una List<Utente> che è mantenuta ordinata tramite l' invocazione del metodo Collections.sort(classifica) in seguito alle operazioni che potrebbero modificarne il corretto ordinamento (operazioni che potrebbero modificare lo score di qualche utente) . Il costruttore della classe inizializza la classifica come una LinkedList<Utente> poi va a vedere se esiste il file con lo stato serializzato della classifica, se sì, effettua la deserializzazione caricando sulla struttura del programma lo stato preesistente. La classe offre dei metodi setter come il metodo add che serve ad aggiungere un Utente alla classifica, il metodo sort viene invocato per tenere ordinata la classifica, oltre a fare l'ordinamento restituisce un booleano che vale true se e solo se è cambiato il podio false altrimenti. Entrando più nel dettaglio del codice l'ordinamento è realizzato dal metodo Collections.sort(classifica), questo metodo effettua l'ordinamento della classifica che è una List<Utente> in ordine crescente secondo l'ordinamento naturale dei suoi elementi (oggetti Utente). Tutti gli elementi della lista implementano l' interfaccia Comparable<> (Essendo una List<Utente> tutti gli elementi implementano l'interefaccia Comparable<Utente> perciò ridefiniscono CompareTo). Inoltre viene memorizzato il podio prima di fare l'ordinamento e il podio dopo aver fatto l'ordinamento questo permette di restituire un valore booleano che vale true se e solo se il podio è variato. Il metodo getPodium restituisce la List<String> che rappresenta il podio, dove ogni giocatore sul podio è identificato dal proprio username.

NotificationService (interface)

Interfaccia del servizio di notifica di variazioni nel podio della classifica, dichiara i metodi che dovranno essere definiti dalle implementazioni di questa interfaccia : registerForCallback, unregisterForCallback e update, rispettivamente per registrarsi al servizio di notifica, deregistrarsi dal servizio di notifica e inviare le notifiche agli iscritti al servizio.

NotificationServiceImpl

Classe che implementa il servizio di notifica attraverso il meccanismo di CallBack RMI. Gli oggetti di questa classe dovranno essere esportati e raggiunti mediante degli stub da remoto, per questo la classe estende RemoteObject. Inoltre dato che questa classe implementa l'interfaccia NotificationService definisce i metodi in essa dichiarati. La variabile privata clients (di tipo List<NotifyEventInterface>) mantiene la lista dei client iscritti al servizio di Callback. Il metodo registerForCallback esposto nell' interfaccia NotificationService prende in input uno stub della classe NotifyEventInterface, stub dell' oggetto remoto che serve per mandare la notifica al client e lo aggiunge alla lista clients. Il metodo unregisterForCallback effettua la deregistrazione del client rimuovendo lo stub dalla lista. Il metodo update invoca il metodo doCallbacks passandogli il nuovo podio che è una List<String> con gli username degli utenti che compongono il podio, ordinata dal primo al terzo. Il metodo doCallbacks iniziallizza un Iterator<NotifyEventInterface> i sulla

lista clients contente gli stub degli oggetti remoti di tipo NotifyEventInterface e con un ciclo while scorro l'iteratore e chiamo il metodo notifyEvent di ciascun oggetto remoto per inviare le notifiche ai vari client.

RegistrationService (interface)

RegistrationService è l'interfaccia del servizio di registrazione di Wordle. Le classi che implementeranno questa interfaccia dovranno definire i metodi in essa esposti, ovvero dei metodi per accedere e manipolare il database delle registrazioni del programma. RegistrationService estende Remote perché le implementazioni di questa interfaccia saranno oggetti remoti i cui metodi potranno essere invocati da remoto secondo il meccanismo di Java RMI.

RegistrationServiceImpl

Classe che implementa il servizio che gestisce il database delle registrazioni, implementa l'interfaccia RegistrationService quindi ne definisce i metodi. Le registrazioni sono mantenute in una ConcurrentHashMap<String, Utente> RegistrationDB, dove la stringa username essendo univoca all' interno del database fungerà da chiave, la ConcurrentHashMap essendo una struttura sincronizzata mi esonera dalla sincronizzazione esplicita. La variabile private c contiene un riferimento ad un oggetto di tipo Classifica. Il costruttore della classe inizializza il database caricando da un file sulle struttura RegistrationDB lo stato serializzato del programma se questo esiste. I metodi, tutti sincronizzati, che la classe mette a disposizione sono il metodo addRegistration che effettua la registrazione di un Utente con username e password come quelli passati in input, il metodo restituisce true se e solo se l' operazione è completata con successo false se si è verificata una condizione che ha impedito il completamento della registrazione come ad esempio se la password immessa era la stringa vuota oppure se il database conteneva già un utente con quell'username. Il metodo setLogged prende in ingresso username e password e setta l'utente avente quell' username come loggato a meno che questo non sia presente nel db delle registrazioni oppure sia presente e già loggato, oppure la password passata al metodo non corrisponda a quella associata all 'utente con quell' username nel db, se mi trovo in uno di questi ultimi casi il login non è effettuato e viene restituito false. Il metodo setUnlogged si comporta sulla falsariga del metodo setLogged eseguendo però il logout anziché il login. La classe mette a disposizione anche altri metodi getter per reperire le variabili locali private come getUser, getPassword, isRegistered, getClassifica e getRegistrazioni.

WordleServerMain

La classe WordleServerMain contiene il metodo main attraverso il quale viene mandato in esecuzione il server di Wordle. Il metodo main di questa classe si incarica anche di andare a leggere i parametri di configurazione del server dal relativo file di configurazione.Nel metodo main in si istanzia un oggetto server della classe WordleServer a cui vengono

passati i relativi parametri di configurazione e poi viene invocato il metodo start con cui viene fatto partire il server.

WordleServer

Classe che implementa il server. Mantiene riferimenti al servizio di registrazione (RegistrationServiceImpl **RegistrationService**), al threadpool che eseguirà i task (ExecutorService **service**), al gioco (Game **game**), alla classifica (Classifica **classifica**) e al servizio di notifica (NotificationServiceImpl **server**), all' indirizzo del gruppo di multicast (InetAddress **multicastGroup**) e alla porta associata all' indirizzo multicast (int **port**). Il metodo start è il metodo attraverso il quale viene messo in stato di esecuzione il server, per prima cosa nella variabile procs salvo il numero dei processori disponibili alla JVM e creo un FixedThreadPool con un numero fisso di thread uguale al valore di procs. Creo la classifica, creo il RegistrationService a cui passo attraverso il costruttore la classifica appena creata ed il servizio di notifica. A questo punto creo il game al cui costruttore passo i riferimenti alla classifica ed al servizio di notifica. Creo gli stub per il RegistrationService e per il NotificationService, ovvero per gli oggetti che andranno esportati. Successivamente creo un registrtry e pubblico i due stub nel registry rispettivamente con i nomi REGISTRATION-SERVICE e NOTIFICATION-SERVICE. Successivamente setto un indirizzo IP ed una porta per il gruppo di multicast, creo e starto il thread che si occuperà di persistere lo stato del gioco, al costruttore della classe Thread passo la runnable serializzatore_stato_server. La classe **serializzatore_stato_server** che implementa l'interfaccia runnable realizza il task che si occupa della serializzazione dello stato del server, mantiene un riferimento al database delle registrazioni (RegistrationService db), questa componente contiene la struttura da persistere e una variabile intera millis che mi dice ogni quanti millisecondi serializzare. L'override del metodo run definisce il task, dormo per millis millisecondi, mi sveglio, creo un'istanza della Classe file, a cui passo la stringa che identifica il path nella directory di lavoro corrente dove risiederà o verrà creato se non esiste il file LogRegistrazioni.json e che conterrà gli oggetti Json serializzati. Creo un FileOutputStream per scrivere sul file. Lo avvolgo in un OutputStreamWriter, faccio il toJson della struttura che devo persistere, che non è altro che la Map<username, Utente> che contiene le registrazioni, ottenendo delle stringhe json che scrivo nel file json. A questo punto predispongo il server per ricevere e gestire le connessioni coi vari client. Dichiaro un ServerSocketChannel **serverChannel** che sarà il mio listening socket per le richieste di connessione provenienti dai client, dichiaro un Selector **selector** che servirà per il multiplexing dei canali. Attraverso il metodo ServerSocketChannel.open() apro un server-socket channel per un IP socket. Il socket del channel appena creato è inizialmente unbound, bisognerà fare il binding ad un indirizzo specifico prima che le connessioni possano essere accettate. Metto quindi in ascolto il listening socket in localhost sulla porta passatagli dal file di configurazione e configuro il channel a non bloccante. Apro il selettore e registro il ServerSocketChannel serverChannel sul selettore selector per le operazioni di ACCEPT. A questo punto il server entra in un ciclo infinito dove il selettore monitorerà i channel a lui registrati. Faccio una selector.select(),

prendo il set delle chiavi dei channel pronti e lo metto nella variabile Set<SelectionKey> readyKeys, prendo un itetatore su questo set e ciclo finchè ci sono elementi nell' iteratore. In SelectionKey key prendo l'elemento che mi fornisce l'iteratore attraverso il metodo next() e attraverso il metodo remove rimuovo la chiave dal SelectedSet ma non dal Registered set. Se la chiave del canale, key, è pronta per accettare una nuova connessione socket, prendo il channel associato alla chiave (che è un ServerSocketChannel), faccio un' accept su quel server socket che mi restituirà il socketchannel per la comunicazione col client, configuro il socketChannel appena creato a non bloccante e lo registro sul selettore per le operazioni di lettura. Per supportare il protocollo di comunicazione che prevede l'invio di un intero e di un messaggio dal client al server creo un array di ByteBuffer, il primo bucket è un ByteBuffer con lo spazio necessario a memorizzare un intero, il secondo bucket è un ByteBuffer con abbastanza spazio per contenere il messaggio, questo array di ByteBuffer lo passo come attachement al momento della registrazione per operazioni di lettura del SocketChannel al selettore. Se la chiave del canale key è pronta per operazioni di lettura, devo poter leggere senza bloccarmi, per far ciò stabilisco un protocollo di comunicazione, il client invia al server prima un intero che indica il numero di byte dal leggere nel messaggio che seguirà e poi invia un messaggio al server nel formato seguente : username/password/cmd/valore. Recupero il SocketChannel associato alla chiave, recupero l'array di bytebuffer che sta nell'attachement, leggo dal socket channel sull'array di Bytebuffer in maniera non bloccante. Se nel primo ByteBuffer ho letto tutti i byte che costituiscono un intero, mi preparo a leggere dal buffer facendo una flip e prendo l'intero l che mi dice quanti byte dovrò leggere nel messaggio che segue. Se la position del secondo bytebuffer ovvero quello in cui andrò a leggere il messaggio sarà pari a l vuol dire che ho letto tutta la richiesta del client, tokenizzo la stringa della richiesta e acquisisco i campi che so essere presenti perché ho stabilito io il protocollo. Prendo quindi username , password, comando. Faccio uno switch case sul comando per andare a riconoscere che tipo di richiesta ha inviato il client e a seconda del tipo di comando inviato nella richiesta faccio eseguire al threapool un opportuno task per gestire tale richiesta passando al costruttore del task i parametri necassari a gestire la richiesta.

Una rapida occhiata ai vari task :

- Il task shareSuggestion si occupa di condividere i suggerimenti sul gruppo di multicast. Il task è definito nella riscrittura del metodo run(), per prima cosa devo rispondere al client per dirgli che ho preso in carico la sua richiesta, questa risposta viaggerà sulla connessione TCP prendo il SocketChannel associato alla SelectionKey key, preparo il messaggio da inviare in cui metto come codice contenente l'esito della richiesta la stringa ok, lo metto in un ByteBuffer, invio il tutto. A questo punto creo un DatagramSocket, prendo la partita relativa all' Utente con l' username giusto, preparo un datagramma in cui metto il messaggio da condividere, la sua lunghezza, con IP e porta del gruppo di multicast e spedisco sul Datagramsocket.
- Il task guess rappresenta il tentativo di un utente di indovinare la secret word. Questo metodo per prima cosa controlla che l'utente che ha inviato la guessed word

stia effettivamente giocando una partita nella sessione corrente del gioca, poi controlla se l'utente ha esaurito i tentativi a sua disposzione, controlla se la parola guessed word inviata sia presente nel dizionario. Se la gw è presente nel dizionario nella variabile suggestion salvo la stringa con i suggerimenti, poi testo se ho indovinato la parola se il test da esito positivo il codice di ritorno è "ok" altrimenti se non ho indovinato ed ho esaurito i tentativi il codice di ritorno è "fine" , se non ho indovinato ma non ho esaurito i tentativi il codice di risposta è "ritenta", se la parola non era lunga 10 caratteri oppure era lunga 10 ma non presente nel dizionario il codice di ritorno è "rinvia" e non viene bruciato nessun tentativo a disposzione, se i tentativi sono esauriti il codice di ritorno è ancora "fine", infine se l'utente non sta giocando nessuna partita nella sessione corrente del gioco il codice è "gioca". A questo punto preparo il messaggio da inviare al client che conterrà il codice con il risultato dell' invio della gw e con la stringa dei suggerimenti, se il codice è "fine" oppure "ok" la partita è terminata perciò invio la traduzione della parola segreta, invio il tutto al client sulla connessione TCP.

- Il task gioca serve per avviare una partita nella sessione corrente del gioco. Viene controllato se l'utente identificato dall' username ha già una partita in corso o termianta per la sessione corrente del gioco, se non la ha viene aggiunta. Viene inviato un messaggio sulla connesione tcp al client per rendere conto dell' esito della richiesta di avvio della partita.
- Il task logout gestisce la richiesta di logout di un Utente. Eseguo l'unlog di un utente chiamando l'opportuno metodo fornito dal servizio di registrazione RegistrationService
- Il task sendStatistics gestisce l'invio delle statistiche di un utente sulla connesione TCP.
- Il task sendRanking gestisce l'invio della classifica sulla connessione TCP. Da notare che poiché la classifica è un' entità che cresce dinamicamente, prima invio un intero che codifica la lunghezza della stringa che contiene la classifica, poi invio la stringa contenente la classifica.

Chiarimenti sul protocollo di comunicazione tra client e server

I messaggi che si scambiano il client e il server sulla connessione TCP seguono un formato ben preciso. Il formato dei messaggi cambia a seconda del fatto che il messaggio sia una richiesta del client diretta al server oppure una risposta del server spedita al client.

I messaggi preparati dal client e inviati al server sono fatti nel seguente modo: viene prima inviato un intero che codifica la lunghezza espressa in numero dei byte del messaggio che segue, il messaggio ha il formato username/password/cmd[/valore], dove username è l'username dell' utente che sta inviando la richiesta così come password è la sua password, cmd è il comando/o tipo di richiesta inviato dal client al server e può essere ad esempio share, per richiedere la condivisione dei suggerimenti sul gruppo multicast, logout per richiedere il logout dell' utente, gioca per richiedere l'avvio di una nuova partita, guess per l'invio di una guessed word, classifica per la richiesta di visualizzare la classifica

e statistics per la richiesta di visualizzare le statistiche. Il campo valore tra parentesi quadre , valore, è opzionale e viene usato per immagazzinare la guessed word in un invio di parola, ovvero quando il campo cmd è guess.

I messaggi preparati dal server in risposta alle richieste del client seguono il seguente formato : username/cmd/codice[/valori_opzionali]/n, dove username è l'username del utente che aveva inviato la richiesta a cui si sta rispondendo, cmd è il tipo di richiesta inviata dal client a cui si sta rispondendo (guess, logout, share, gioca, statistics, classifica), il codice indica l'esito della richiesta inviata dal client , nella maggior parte dei casi 'ok' significa che la richiesta è andata a buon fine, 'no' vuol dire che non è stato possibile eseguire la richiesta, poi ci sono una serie di valori opzionali che sono presenti a seconda del tipo di risposta per esempio la risposta di tipo guess può avere come valori opzionali la stringa con i suggerimenti da inviare al client oppure in taluni casi (rappresentati da particolari codici) la traduzione della parola segreta ed infine il carattere '/n' perché server deve inviare delle linee, stringhe terminate da '/n', il client utilizza questo carattere di terminazione per fare il parsing. La risposta ad una guess prevede dei codici un pochino più raffinati ad esempio 'ok' codifica la vittoria della partita , parola indovinata, 'fine' parola non indovinata ma ancora tentativi a disposizione, 'rinvia' parola da rinviare perché non presente nel dizionario, eccetera …

Schema generale dei thread attivati lato server e lato client

I thread attivati lato client sono due il main thread e il thread che gestisce l'iscrizione al gruppo di multicast e che sta in ascolto continuamente per leggere i messaggi inviati sul medesimo gruppo, questo thread denominato gestoreMulticast è creato e startato dal main thread del client dopo l'avvenuto login di un utente.

Lato server, oltre al main thread del server, si contano il thread per la serializzazione dello stato denominato serializer che viene creato e startato nel main thread del server nel metodo start al momento dell'avvio del server, il thread per il refresh del gioco , quindi la chiusura della sessione corrente, con la terminazione delle partite in atto e l'avvio di una nuova sessione, con una nuova parola segreta e le strutture della sessione resettate. Questo thread è creato e startato nel metodo costruttore della classe game, che viene invocato anch' esso al momento dell' avvio del server. Sono presenti lato server anche i thread del FixedThreadPool creato anch' esso al momento dell' avvio del server. I thread del pool sono in numero fisso e pari al numero di core della macchina.

Per essere precisi, sia lato client quando viene esportato l'oggetto della classe NotifyEventImpl sia lato server quando vengono esportati gli oggetti delle classi RegistrationServiceImpl e NotificationServiceImpl vengono creati e startati appositi thread per far si che che gli oggetti remoti possono essere in ascolto delle chiamate dei loro metodi in arrivo da remoto.

Descrizione delle primitive di sincronizzazione utilizzate per accedere a strutture dati condivise

Per garantire che il programma sia thread-safe, le strutture condivise dai thread sono sincronizzate attraverso il costrutto synchronized offerto da Java usato a livello dei metodi per le strutture critiche. In particolare le classi Game, Utente, Statistiche, Partita, RegistrationServiceImpl, NotificationServiceImpl le cui istanze possono essere risorse condivise fra i thread del programma, dando luogo a possibili race conditions dovute all' accesso concorrente, dichiarano sia i metodi getter, sia i metodi , sia altri metodi di supporto che possono risultare critici, come synchronized. Questo meccanismo assicura l'accesso dei thread a queste risorse in mutua esclusione. La controindicazione dell' utilizzo di questo modificatore è che è necessario sincronizzare sia i metodi che modificano la risorsa sia quelli che la leggono, vale a dire che un thread che esegue un metodo su quella risorsa che è un metodo di sola lettura esige la mutua esclusione, un altro thread che vuole leggere la medesima risorsa rimane fuori in attesa che si liberi la lock implicita dell' oggetto su cui lavorano i metodi synchronized.

Alcune strutture come il database delle registrazioni RegistrationDB del server che contiene le associazioni tra username e utente con quell' username o il database della sessione del gioco, la struttura players che contiene le associazioni tra utente e partita dell' utente in corso in quella sessione sono implementate come ConcurrentHashMap la prima ConcurrentHashMap<String, Utente> la seconda come ConcurrentHashMap<Utente, Partita>, perché data la possibilità che queste risorse siano condivise fra thread e quindi accedute concorrentemente, la ConcurrentHashMap<> offerta da Java nella concurrent collection essendo una struttura sincronizzata mi esenta dalla sincronizzazione esplicita semplificandomi il lavoro. Medesimo ragionamento è stato fatto per il Vector<String> che immagazzina i messaggi letti dal thread lato client che sta in ascolto sul gruppo di multicast, che è anche condivisa dal main thread del client che deve stampare questi messaggi, perciò utilizzo la classe Vectro<E> di java che è sincronizzata e thread e safe a differenza ad esempio di ArrayList<E> che non lo è.