Laboratorio di Reti
WORDLE: un gioco di parole 3.0
Progetto di Fine Corso A.A. 2022/23

Contenuti:

Struttura del progetto

Client:

- **WordleClientMain**

  La classe principale del client, contiene tutte le operazioni che il client svolge, ad esclusione di quelle specificate in seguito.
  Si occupa quindi in primo luogo di caricare i parametri dal file client.properties, inizializzare TCP e RMI, sia lato client per la callback che collegarsi a quello del server. Si occupa di avviare poi il thread separato che gestisce il multicast. Procede poi a interagire con l'utente tramite un menu iniziale che consente registrazione, login o chiusura dell'applicazione. Se si sceglie l'opzione di login, si accede al vero e proprio gioco, contenente un altro menu con tutte le opzioni necessarie: fare una partita, visualizzare le statistiche, visualizzare i messaggi arrivati sul multicast, visualizzare la leaderboard corrente o fare il logout.

- **ClientMulticastHandler**

  È una classe che implementa runnable. Il main genera un thread all'avvio e rimane attivo fino alla chiusura del client. Resta in attesa di messaggi sulla rete multicast e alla ricezione provvede a inserirli in una cache implementata come ConcurrentLinkedQueue<String>. Il ClientMain può accedere a questa cache chiamando un metodo statico implementato in questa stessa classe, che effettua una pop dalla testa della concurrentLinkedQueue e stampa i messaggi ricevuti sulla CLI.

- **RMIClient**

  Gestisce la comunicazione della leaderboard a ogni suo aggiornamento da parte del server via RMIcallback. Riceve una comunicazione iniziale tramite setLB all'instaurazione della comunicazione e la aggiorna ogni volta che viene invocato il metodo notifyEvent da parte del server. Provvede anche ad avvisare il client dell'update stampando un messaggio sulla CLI. Il salvataggio della leaderboard avviene in una ConcurrentLinkedQueue<String[ ][ ]>, rispettando il formato originale della Leaderboard ma garantendo thread safety.

Server:

- **WordleServerMain**

  Il main effettivo del server. Si occupa di caricare i parametri da server.properties, effettuare il setup delle connessioni tcp verso client e multicast, caricare il database degli utenti da file su una ConcurrentHashMap<String,User>, avviare RMI, avviare il thread che genera una parola ogni intervallo fissato dall'utente (parametro in properties), avviare il thread che aggiorna periodicamente il file su disco e avviare la threadpool che gestirà le connessioni con i vari client.
  Il ServerMain si arresta tramite SIGINT, che viene catturato tramite shutdownHook.

- **ShutdownListener**

  Classe che estende Thread, collegata allo ShutdownHook. Quando riceve il segnale di arresto, salva il database su file e effettua l'arresto del server.

- **ServerTask**

  Classe che implementa runnable. È il cuore vero e proprio del server, poiché ogni istanza di ServerTask si occupa di comunicare con un client connesso separatamente.
  Per prima cosa instaura la connessione e si connette al clientRMI per la callback, aspettando che il client decida di fare il login. Dopo aver effettuato il login correttamente, resta in attesa dell'opzione del menu scelta dal client e implementa singolarmente le operazioni necessarie, a volte chiamando funzioni di altre classi.

- **GameFunctions**

  Raccolta di funzioni che gestiscono il gioco effettivo.
  Rilevante è la funzione playTurn: gestisce un turno di gioco effettivo. Riceve una parola e la parola corretta per quella partita, controlla lettera per lettera se sono uguali e in caso negativo controlla successivamente se vi sono occorrenze di quella lettera che non sono già state segnate come uguali. Invia al client un ByteArray contenente una sequenza che indica il colore di ogni lettera.

- **DBUpdater**

  Classe che implementa Runnable, si occupa di aggiornare periodicamente il file su disco.

- **Leaderboard**

  Classe che gestisce la leaderboard e raccoglie metodi synchronized per l'accesso concorrente, siccome una matrice non è una struttura Thread-safe. Contiene loadLB, che si occupa di generare la leaderboard direttamente dalla concurrentHashMap, e updateLB, chiamata ogni volta che una partita si conclude.

- **RMIServer**

  Gestisce la registrazione dei nuovi utenti alla piattaforma. Restituisce al client un messaggio di errore per formato non corretto o utente già esistente, altrimenti un messaggio di conferma.

- **WordsManager**

  Classe che implementa runnable. Si occupa di estrarre una parola ogni intervallo di tempo configurato. Contiene anche un metodo chiamato all'avvio che carica le parole dal file su disco in un HashSet<String>

- **User**

  La classe che implementa l'oggetto User vero e proprio. Contiene tutti i parametri relativi all'utente, quali username, password, punteggio, numero di vittorie e sconfitte e la distribuzione di esse, oltre che la serie di vittorie corrente e massima.

  Contiene inoltre tutti i metodi per ottenere e impostare i parametri, chiamati dalle varie funzioni nel programma.
  - Ho effettuato una modifica al punteggio: la formula utilizzata è:
    (media di punti a partita*nr vittorie) - nr sconfitte,
    dove i punti a partita assumono un valore da 6 a 0.5, con scalini di 0.5, a seconda del numero di tentativi impiegati per trovare la parola
  - In questo modo **si premia chi trova una parola in meno tentativi e si penalizzano le sconfitte**, di conseguenza gli utenti possono perdere posizioni in classifica e aumenta la competitività.

Strutture dati

- **Client.properties**

  Contiene i parametri configurabili per il client, cioè: IP del server, porta del server, IP del multicast, porta del multicast.

- **Server.properties**

  Contiene i parametri configurabili per il server, cioè: IP del server, porta del server, IP del multicast, porta del multicast, porta di RMI, il tempo di validità di una parola in secondi

- **Words.txt**

  Fornito da consegna, file con il dizionario delle parole.
  - Viene caricato in memoria in un HashSet in modo da avere tempo O(1) per l'accesso alla parola per controllare l'esistenza nel dizionario.
    Lo spazio è O(n) ma è ottimizzato rispetto a una ricerca binaria direttamente su file per via della cache della CPU.
  - La parola corrente invece viene selezionata accedendo direttamente al file tramite un RandomAccessFile, in modo da poter direttamente prendere una riga casuale. In O(1) spazio e tempo

- **DB.json**

  Tiene in memoria il database degli utenti in formato JSON. Caricato sul programma in una concurrentHashMap.

Altre risorse

- **idle.wav, game.wav, win.wav, lose.wav, SE.wav** [1]

  File musicali caricati durante le varie sezioni di gioco, rispettivamente:
  - Idle: musica del menu di gioco
  - Game: musica del gioco
  - Win: Sound effect di vittoria
  - Lose: Sound effect di sconfitta
  - SE: Sound effect generico, utilizzato solo al logout.

[1] L'applicazione è strutturata in modo da avviarsi anche se questi ultimi mancano. Naturalmente in quel caso non vi è musica.

Struttura dei Thread

Client:

ClientMain avvia:
- Un thread MulticastHandler all'avvio, per la ricezione dei messaggi in multicast.

Server:

ServerMain avvia:
- Un thread di tipo wordsManager scheduledAtFixedRate che aggiorna la parola corrente ogni intervallo di tempo
  - Precisamente uso una scheduledThreadpool di size 1
- Un thread di tipo DBUpdater per l'aggiornamento del file del database
- Una cachedThreadpool che lancia thread di tipo ServerTask ogni volta che un client richiede la connessione.
  - Uso cachedThreadpool perché mi permette di ottimizzare riutilizzando Thread inattivi

Thread Safety

Client:

messageCache in MulticastHandler e leaderboard (board) in RMIClient: Uso una ConcurrentLinkedQueue per renderle thread-safe.

Server:

DB: utilizzo una concurrentHashMap che è thread-safe, quindi tutte le operazioni sul DB avvengono thread-safe
Leaderboard: utilizzo metodi synchronized, rendendo l'accesso thread-safe
Hashset<words> in WordsManager: ha accesso read-only dopo il caricamento iniziale, quindi è thread-safe

Protocollo di comunicazione

La comunicazione effettiva avviene tra il ClientMain e l'istanza di ServerTask generata dal ServerMain all'accettazione della comunicazione.

Escludendo l'opzione di register, che invoca una funzione apposita sul server RMI che restituisce una stringa contenente il messaggio da visualizzare su CLI, la comunicazione tra ClientMain e ServerTask avviene a partire dal login.

Per prima cosa il client invia un messaggio contenente [*user*\n*password*], precedentemente sanificati. Il server provvede a estrarre le due informazioni comunicate e a fare i controlli necessari per login. Risponde con un messaggio di errore del tipo [*errore*\n] in caso il login non vada a buon fine, altrimenti risponde con [*user*].

Una volta effettuato il login correttamente, l'utente può mandare diversi tipi di messaggi a seconda delle opzioni del menu, che descrivono diversi casi d'uso:

- **Caso [playGame]:**

  Il server controlla se l'utente ha già effettuato una partita per la parola corrente e a seconda del caso invia il messaggio d'errore [cantPlay\n] o di conferma [canPlay\n].

  Nel caso di conferma inizia uno scambio di messaggi tra client e server, in cui il client invia una parola di 10 lettere e il server in primo luogo controlla se è una parola del dizionario, inviando [notValid\n] in caso non lo sia. Altrimenti passa al confronto con la parola corrente, inviando [equal\n] in caso abbia indovinato la parola e [notEqual\n] in caso sia errata, seguito da un byte[10] contenente lo schema degli indizi che viene poi elaborato lato client.

  Se sono stati effettuati i 12 tentativi, il server invia un messaggio [*correctWord*] contenente la parola corrente. Dopodiché, sia in caso di vittoria o di sconfitta, il client invia un messaggio [notShare\n] se decide di non condividere il risultato sul multicast, altrimenti invia [share\n] e il server condivide un datagramma con il risultato sul multicast, inviando un messaggio di conferma [done\n] al client.

- **Caso [sendStatistics]:**

  Il server invia una sequenza di messaggi, rispettivamente: 13 interi singolarmente per la winDistribution in ordine, un intero per il numero di vittorie, un floating-point per il punteggio, un intero per la streak di vittorie corrente, un intero per la streak massima.

- **Caso [logout]:**

  Una volta ricevuto, il server si occupa di effettuare il logout impostando User.setLoggedIn(false) e inviando [completed\n] come conferma del login.

Le altre opzioni del menu chiamano funzioni interne al client (rispettivamente *ClientMulticastHandler.publishAll( )* per stampare i messaggi ricevuti sul multicast e *printLb( )* per stampare la leaderboard corrente)

Immagini di gioco

```
Loading game...
Hello a, welcome to 10WORDLE! What do you want to do today?
1. Play game   2. Show statistics   3. See multicast shared messages
4. See leaderboard   0. Logout
```

1. Menu di gioco

```
Input your guess. Tries left: 12
didactical
Word not correct. Here is how you did:
didactical
Input your guess. Tries left: 11
tooshort
Please input a 10 letter long word (does not count as a try).
Input your guess. Tries left: 11
notdiction
Please input a valid word (does not count as a try).
```

2. Esempio di gioco

```
Input your guess. Tries left: 10
taxidermic
The leaderboard has been updated! Go see the new leaders and their scores.
Correct!
You won the game guessing the word "TAXIDERMIC" in 3 tries.
Here's the italian translation of the current word: tassidermico
Do you want to share your results? Y/N
Y
The result was successfully shared!
```

3. Partita vinta

```
Getting the game statistics for a...
Total matches: 14    Wins: 7    Losses: 7
Win percentage: 50.00%
Current score: 11.08
Current win streak: 0   Max win streak: 6
Win distribution for number of tries:
1: 4   2: 0   3: 0   4: 0   5: 0   6: 1
7: 0   8: 1   9: 0   10: 0   11: 1   12: 0

Generating win distribution graph...
1: ##########################################################
2:
3:
4:
5:
6: ###############
7:
8: ###############
9:
10:
11: ###############
12:
Done!
```

4. Statistiche di gioco

```
Choose an option: 1. register    2. login    0. exit
1
Insert Username:
Esem
Insert Password:
Pio
Operation complete
```

5. Registrazione alla piattaforma

Come compilare

Posizionarsi nella cartella del progetto e eseguire i comandi:

- mkdir build

  Crea la cartella dove andranno i pacchetti compilati

- javac -d build -cp lib/gson-2.10.1.jar src/Server/*.java src/common/*.java

  Compila nella cartella build il pacchetto Server usando la libreria esterna gson

  - javac: eseguibile del compilatore del jdk (java development kit)
  - -d build: directory di destinazione del pacchetto
  - -cp lib/gson-2.10.1.jar: classpath per librerie esterne, in questo caso gson
  - src/Server/*.java src/common/*.java: cosa compilare

- javac -d build -cp lib/gson-2.10.1.jar src/Client/*.java src/common/*.java

  Stesso comando di prima ma per il client

Come eseguire [2]

- Server:

  java -cp lib/gson-2.10.1.jar:build Server/WordleServerMain

  - java: strumento di esecuzione di file java compilati
  - -cp lib/gson-2.10.1.jar:build: classpath, quindi la libreria gson nella cartella lib e la cartella con il pacchetto contenente tutti i file .class generati (su Windows le classpath dovranno essere separate da , invece che da :)
  - Server/WordleServerMain: percorso per la classe contenente il main, non è necessario indicare il .class

- Client:

  java -cp lib/gson-2.10.1.jar:build Client/WordleClientMain

[2] assicurarsi che i file client.properties, server.properties, words.txt e gson-2.10.1.jar siano presenti nelle directories corrette.