Relazione Progetto di fine corso Laboratorio III

Filippo Fornai

WORDLE: un gioco di parole 3.0

CONSIDERAZIONI GENERALI

Il sistema è stato sviluppato tenendo conto di solo e unicamente eventuali errori di tramissione dal server al client. Qualsiasi errore di questo tipo renderebbe inconsistente lo stato del server rispetto alle azioni del client.

Il sistema è stato sviluppato interamente su Java 8.

Come da specifica i parametri, sia del server che client, vengono letti in automatico da due file json, rispettivamente "***ServerSettings.json***" e "***ClientSettings.json***", che sono letti attraverso un metodo chiamato "loadSettings" e consecutivamente salvati all'interno di proprietà statiche. Se i file sono assenti o non sono formattati adeguatamente viene terminata l'esecuzione del programma (sia nel caso del server che del client). Inoltre se è presente il file "***words.txt***" ma non sono contenute parole al suo interno il server termina. E' necessaria la presenza di almeno una parola per il gioco, tutte le parole devono avere una newline ("\n") dopo e le parole devono essere ordinate lessicograficamente.

La classe Stat (illustratata nella sezione SERVER) è comune a entrambi i programmi per operare serializzazione e deserializzazione in modo efficiente durante la condivisione dei dati.

Per evitare ambiguità nelle seguenti spiegazioni si fa riferimento ai thread con i nomi dei task che eseguono (e.g. si chiama Client thread il thread che esegue la task Client che implementa Runnable)

CLIENT

Threads

Il client è composto da due thread durante l'esecuzione, il Main thread e il MulticastListener thread. Il primo viene eseguito all'esecuzione del processo, mentre il secondo viene eseguito "on-demand" come descritto in seguito.

Descrizione generale flusso di esecuzione e considerazioni

Dopo l'inizializzazione dei parametri descritta in "**Considerazioni generali**" il Main thread si occupa di interagire con l'utente chiedendo quale delle diverse azioni vuole eseguire e guidandolo in caso abbia scelto un comando che richiede più step per essere completato.

Ogni comando che è stato accettato di eseguire dal client viene mandato al server tramite un metodo chiamato "***user_server_response()***". Il server risponderà con un codice che indicherà l'accettazione o il rifiuto del comando e il motivo per cui è stato rifiutato.

Si definisce "sessione" il periodo che intercorre tra un "**login**" e un "**logout**" avvenuti con successo. Ogni client può avere più sessioni.

La connessione tcp tra il client e il server viene instaurata quando viene eseguito un **“login”** o un **“register**” attraverso il metodo “**connect()**” e persiste fino alla chiusura del processo. Nel caso in cui la connessione cada o venga chiusa da parte del server il client rimane attivo e viene chiamato il metodo “**disconnect()**”, che rende nulli tutti i riferimenti ai socket tcp precendetemente creati.

La connessione udp che verrà utilizzata per lo sharing sarà gestita dal MulticastListener thread, che verrà avviato solo all’avvenuto login, da quel momento in poi continuerà ad ascoltare per risultati condivisi fino alla terminazione del processo. A differenza dei socket tcp che vengono resi nulli in caso di caduta/chiusura dell connessione, il socket multicast rimane attivo fino alla terminazione del processo.

Strutture dati condivise

I due thread condividono:

- **Le proprietà “MULTICAST_IP” e “MULTICAST_PORT**” che vengono inizializzate dal Main e poi lette dal MulticastListener (rese volatile per far in modo che l’inizializzazione sia stata propagata in memoria centrale al momento della lettura del MulticastListener threads).
- **La struttura condivisa**
  ```
  final static LinkedBlockingQueue<String> shared_results = new LinkedBlockingQueue<>();
  ```
  in cui vengono salvati i risultati condivisi dal MulticastListener e letti i risultati dal Main attraverso il metodo:

  ```
  public static void show_shared_result(){. . .}
  ```
  (che è usato solo dal Main)

  I due metodi al loro interno sono sincronizzati su “**shared_results**” per assicurare la thread safety.

SERVER

Threads

Il server è composto da 4 classi di thread durante l’esecuzione:

- **Main**: eseguito per primo, si occupa di vari aspetti del server:
  - legge e salva i parametri di configuazione presenti in “**ServerSettings.json**”, carica tutte le parole che potranno essere usate per il gioco dal un file chiamato “words.txt” e carica i dati dei giocatori dal file “Data.json” (tutti questi dati saranno poi salvati in strutture dati condivise che verranno trattate nelle sezioni seguenti).
  - si occupa di creare l’ “accept” socket dove si connetteranno i client e il multicast socket dove verranno condivisi i risultati dai giocatori.

    - Crea 1 **“WordChanger**” thread gestito da uno “**ScheduledExecutorService**”, 1 “**ServerEnder**” thread e una “**FixedThreadPool”** (tutti descritti dopo il Main)
    - Accetta le connessioni in entrata passandole ai thread della **FixedThreadPool**
    - Nel momento dell’arrivo del comando “**exit”** da linea di comando si occuperà della chiusura dei socket, dei thread e threadPool presenti e del salvataggio dei dati precedentemente caricati e aggiornati dalle sessioni di gioco avvenute.

- **ServerEnder**: lo scopo di questo thread è unico, ricevere da linea di comando il comando **“exit**”. Nel caso in cui venga usato il comando il thread si occupa di cambiare una proprietà volatile chiamata “**end**”, che segnala a tutti i thread che è stata richiesta la terminazione del processo.

- **WordChanger**: lo scopo di questo thread è cambiare la cosìdetta “parola del giorno” (**volatile string wordOfTheDay**). La presenza di un thread che cambia in modo attivo la parola del giorno e la presenza di thread che accedono in lettura questa proprietà ha reso necessario delle ulteriori considerazioni sulle race conditions. Dato il seguente contesto:
    - La parola del giorno è inizializzata in modo statico come una stringa vuota (“”):
    - Viene eseguito WordChanger;
    - Viene istantaneamente deschedulato senza cambiare la parola del giorno;
    - Un thread creato dopo la creazione di WordChanger accede alla parola del giorno (che è rimasta nulla data la deschedulazione di WordChanger);

  Per evitare lo scenario la parola è inizializzata dal Main prima che siano presenti possibili thread che vogliono accedere alla parola, e l’esecuzione del WordChanger viene schedulato con un delay RESET_TIME (dato che la parola sarà già stata inizializzata dal Main)

- **Client**: thread che fanno parte della “**FixedThreadPool”** citata precedentemente. Si occupano a tutti gli effetti di gestire le richieste dei client. Ogni thread gestisce un singolo client e rimane attivo fino alla chiusura della connessione tcp assegnata ad esso.

Descrizione generale flusso di esecuzione e considerazioni

Main

Dopo l’inizializzazione descritta precedentemente il compito del main è effettivamente solo accettare le connessioni in entrata e passare i nuovi socket (dati dall’accetazione) a uno dei thread della “**FixedThreadPoo**l”. In seguito, arrivato il comando “**exit**”, si preoccuperà di eseguire le dovute procedure di chiusura.

ServerEnder e **WordChanger**

Vengono eseguiti nel Main e svolgono i compiti descritti nella sezione precedente fino a terminazione.

Client

I Client thread, dopo essere stati avviati dal Main, si occupano di gestire tutte le richieste dei client. Un Client thread termina solo e unicamente quando la connessione con il client assegnato viene chiusa. Di conseguenza, ogni Client thread potrà gestire più utenti e diverse sessioni di gioco. In modo simmetrico al Client, il Client thread gestisce le richieste con il protocollo precedentemente descritto **(user_server_response -> accettazione/rifiuto comando -> esecuzione comando)**

Strutture dati condivise

Static class Stat: come precedentemente illustrato, classe di supporto che descrive le statistiche di un giocatore

Static class Player: classe di supporto che contiene la password di un qualsiasi utente e le statistiche associate ad aesso

Static class Database: tutte le proprietà della classe sono private e di conseguenza l'unico modo di modificarle o usarle è attraverso i metodi che la classe fornisce. Le proprietà sono le seguenti:

- ***I dati dei giocatori (username, password, statistiche):***
  I dati sono salvati all'interno di una ConcurrentHashMap chiamata "**data**".

```
private static ConcurrentHashMap<String,Player> data;
```

  La scelta della struttura dati è stata guidata in modo tale che le ricerche degli username avvenissero in tempo log(1) (essendo un hashmap) e in modo tale che accessi concorrenti potessero essere eseguiti in parallelo.
  All'interno di "**data**" **a ogni username è associata un oggetto di classe Player**, descritto precedentemente.

  I metodi associati a "data" sono i seguenti:

```
public static String register(Scanner in, PrintWriter out)
```
  usato per registrare un nuovo utente nel sistema se il client non ha già effettuato il login

```
public static String login(Scanner in,PrintWriter out)
```
  usato per controllare la presenza di un utente nel sistema e rendere possibile il login

```
public static Stat get_stat(String username){. . .}
```
  metodo che restituisce la locazione in memoria delle statistiche del Player.

```
public static void lose(String username)
```
  metodo che aggiorna le statische di un giocatore in caso perda

```
public static void win(String username, int tries)
```
  metodo che aggiorna le statische di un giocatore in caso vinca

Nel caso in cui un Client thread debba eseguire il comando "**statistics**" può accedere direttamente all'oggetto Stat attraverso il metodo "**get_stat**". Non sono presenti problemi di sincronizzazione in quanto il sistema è progettato per non permettere due login contemporanei dello stesso account. In questo modo è garantito che a ogni oggetto Stat solo un thread possa avere accesso.

- ***I giocatori online***

```
private static final Set<String> online_players =
Collections.newSetFromMap(new ConcurrentHashMap<>());
```

Implementati come un Set creato da una concurrentHashMap. Rende possibile accessi concorrenti pur mantenendo la logica del Set (ogni player può essere online da un solo client)

- ***I giocatori che hanno già provato a indovinare il wordle prima del reset giornaliero***

```
private static final Set<String> played_players =
Collections.newSetFromMap(new ConcurrentHashMap<>());
```

implementati come un Set create da una concurrentHashMap. Stesse motivazioni di played_players (ogni giocatore può giocare una sola volta al giorno)

Oltre al Database sono presenti altre due strutture e una proprietà condivisa:

- ***Il vocabolario delle parole di wordle e la parola del giorno***

```
static volatile RandomAccessFile vocabulary;
static volatile String wordOfTheDay = "";
```

il vocabolario implementato come un RandomAccessFile in quanto si dovranno fare accessi solo in lettura e molte volte in modo randomico o comunque non lineare, la parola del giorno implementata come una stringa

I metodi associati a "**wordOfTheDay**" sono:

```
public static void changeWOTD(){. . .}
```

per estrarre una nuova parola dal vocabolario e usarla come parola del giorno

```
public static String getWOTD(){. . .}
```

per ottenere la parola del giorno

```
synchronized public static boolean inVocabulary(String word) {. . .}
```
per controllare che la guess di un client mentre si gioca sia nel vocabolario, è implementato con una binary search che opera sul file “words.txt”

- ***I socket di tutti i client attivi***

```
static List<Socket> sockets = Collections.synchronizedList(new ArrayList<>());
```
Per una questione di scelte di implementazioni non è possibile rendere le procedure interne ai Client thread non bloccanti. Di conseguenza, per far in modo che i Client terminino anche se “**bloccati**” su una chiamata di I/O, è stato fatto in modo che il Main abbia accesso a “**sockets**” che contiene tutti i socket accettati di tutti i Client attivi.
Nel momento in cui sarà richiesta la terminazione il Main li chiuderà, generando un eccezione nei Client thread e , di conseguenza, terminandoli.

Struttura del progetto

Il progetto viene fornito sotto forma di archivio zip. È necessario estrarre il contenuto per eseguire i programmi.

All’interno della cartella che si è scelto per estrarre il contenuto dell’archivio saranno presenti due altre cartelle, chiamate rispettivamente “**WordleClient**” e “**ServerClient**”.

A entrambe le cartelle è comune la presenza di “**gson-2.10.1.jar**”, una libreria java che viene usata dal client e server.

All’interno della cartella del **Client** sono presenti:

- **La classa Main del Client chiamata “WordleClientMain”**
- **Il file json contenente i parametri operativi del Client chiamato “ClientSettings.json”.** I valori dei campi del file sono tutte stringhe e sono modificali secondo le seguenti convenzioni:
    - ***“server_ip”:*** rappresenta l’ip del server a cui ci si vuole connettere;
    - ***“server_port”:*** rappresenta la porta del server su cui ci si vuole connettere;
    - ***“multicast_ip”:*** rappresenta l’ip del multicast channel a cui ci si vuole connettere;
    - ***“server_port”:*** rappresenta la porta del dell’host rappresentato da “***multicast_ip***” a cui ci si vuole connettere;

All’interno della cartella del **Server** sono presenti:

- **La classa Main del Client chiamata “WordleServerMain”**
- **Il file json chiamato “ServerSettings.json” contenente i parametri operativi del Server**. I valori dei campi del file sono tutte stringhe e sono modificali secondo le seguenti convenzioni:

    - ***"server_port"***: rappresenta la porta del server su cui si vuole aprire l'accepting socket che accetterà le connessioni in entrata dei client;
    - ***"multicast_ip":*** rappresenta l'ip del multicast channel a cui ci si vuole connettere;
    - ***"multicast_port":*** rappresenta la porta del dell'host rappresentato da "***multicast_ip***" a cui ci si vuole connettere;
    - ***"letters":*** la lunghezza delle parole del vocabolario;
    - ***"max_tries":*** il numero di possibilità che un giocatore ha per indovinare una parola;
    - ***"n_threads":*** il numero di thread disponibili per gestire i client;
    - ***"reset_time":*** il tempo in secondi che intercorre fra i cambiamenti delle parole del giorno.
- **Il file json chiamato "Data.json" contenente i dati dei giocatori** (nel caso in cui sia la prima volta che si usa sarà vuoto)
- **Il file di testo (.txt) chiamato "words.txt" contente le parole usabili per il gioco** (nel caso in cui si scelga di modificare il vocabolario si ricorda che è necessaria almeno una parola per far avviare correttamente il server e ogni parola deve essere seguita da una newline "\n")

NB: a tempo di esecuzione vengono eseguiti dei controlli sui file di supporto, nel caso in cui non vengano rispettate le convenzioni o il format dei file comparirà un warning esplicito.

Valore di default e configurazioni di test

All'interno dei due files di settings i valori dei campi, oltre a poter essere usati come descritto nella sezione precedente, possono essere anche settati a un valore <= "0". Questo valore ha come significato quello di usare un'opzione di "default", hardcoded all'interno del codice. Di seguito si spiega dove è possibile usarlo, cosa implica e in determinati casi quali vincoli impone.

Campi in cui il valore <= "0" è usabile senza vincoli:

- All'interno del server:
    - "**letters**": *default = 5*;
    - "**max_tries**": *default = 6*;
    - "**n_threads**": *default = 3*;
    - "**reset_time**": *default = 86400* (un giorno);

Configurazioni di test

Le seguenti configurazioni di campi sono usabili solo e unicamente come descritto.

La presenza di uno solo dei campi con valore <="0" determina l'uso della configurazione che dovrà essere impostata dall'utente, modificando il resto dei campi presenti nella stessa configurazione a <="0".

Date le precedenti considerazioni vengono solo enumerati i campi che fanno parte della stessa configurazione:

- **Configurazione di test** per **connessione tcp**:
    - Nel **Server**:
        - "**server_port**"
    - Nel **Client**:
        - "**server_ip**"

        - “**server_port**”

Apre il socket del server sulla porta 10000 del proprio ip e fa in modo che il client si colleghi

- **Configurazione di test** per il **multicast channel**:
    - Nel **Server**:
        - **“multicast_ip”**
        - **“multicast_port”**
    - Nel **Client**:
        - **“multicast_ip”**
        - **“multicast _port”**

Apre il multicast socket sulla porta 10000 dell’indirizzo 224.0.0.0, un ip speciale presente sulla propria rete locale. Allo stesso modo il client si unirà al gruppo di multicast presente su quell’ip locale su quella porta.

Esecuzione “automatica”

Nel caso si voglia eseguire il **client** e/o il **server** sarà possibile eseguire i due **“bat”** file presenti nella cartella iniziale, nominati rispettivamente: **“SERVER_START.bat”** e **“CLIENT_START.bat”**.

Compilazione e esecuzione “manuale”

Nel caso si voglia compilare e eseguire manualmente (da linea di comando) i due applicativi seguire la seguente procedura:

- Si apra una console interna alla cartella del programma che si vuole eseguire (o client o server);
- Per compilare:
    - Nel caso in cui si voglia compilare il Client si digiti e si esegua il seguente comando:

      ```
      javac -cp ./gson-2.10.1.jar WordleClientMain.java
      ```
    - Nel caso in cui si voglia compilare il Server si digiti e si esegua il seguente comando:

      ```
      javac -cp ./gson-2.10.1.jar WordleServerMain.java
      ```
- Per eseguire (dopo aver compilato) :
    - Nel caso in cui si voglia eseguire il Client si digiti e si esegua il seguente comando:

      ```
      java -cp ./gson-2.10.1.jar;. WordleClientMain
      ```
    - Nel caso in cui si voglia eseguire il Server si digiti e si esegua il seguente comando:

      ```
      java -cp ./gson-2.10.1.jar;. WordleServerMain
      ```
      (NB: il progetto è stato sviluppato su Windows e di conseguenza questi comandi sono stati utilizzati su “cmd” , nel caso in cui si stia usando MAC-OS o Linux il “;” (punto e virgola) presente dopo “jar” nell’argomento di “-cp” deve essere sostituito da “:” (due punti)

A questo punto il programma scelto sarà in esecuzione.

Descrizione esecuzione Server

Il server durante la sua esecuzione informa l'utente degli step che vengono portati a termine. Una volta inizializzato del tutto informa unicamente di quando sono presenti connessioni in entrata che sono state accettate fino alla richiesta di terminazione che dovrà avvenire digitando il comando "exit" da linea di comando. Ricevuto il comando il server informerà, dopo le dovute procedure, la chiusura del processo e il completamento del salvataggio dei dati dei player, aggiornati dalle sessioni avvenute durante l'esecuzione.

Descrizione esecuzione Client

Il client ha una natura più interattiva rispetto al server. Avviato il programma si viene accolti da un messaggio di benvenuto e una seguente richiesta di scrivere uno dei comandi presenti sullo schermo. I comandi sono:

- **Register**: fa partire una procedura guidata per registrare un nuovo account nel sistema;
- **Login**: nel caso in cui si sia già registrati il comando permetterà di accedere al sistema e al proprio account seguendo le istruzioni che compariranno a schermo;
- **Logout**: nel caso in cui sia già stato eseguito il login all'interno di un account sarà possibile eseguire un logout. Il logout comporta la sola uscita dall'account e non la chiusura del programma, di conseguenza il client potrà eseguire molteplici login durante l'esecuzione del programma;
- **Statistics**: mostra le statistiche dell'account in cui si è fatto il login.
- **Play**: incomincia una nuova partita a Wordle nel caso non si sia ancora giocato prima del reset giornaliero. Durante il gioco sarà possibile sia mandare una propria guess grazie al comando "**send [parola]**" sia fare il logout tramite il comando "**logout**". Nel caso in cui si faccia il logout la partita sarà considerata persa e influirà sulle statistiche. Nel caso in cui la parola del giorno cambi durante una partita ci sarà la possibilità di rigiocare dopo aver completato la sessione corrente.

  Per ogni parola mandata come guess il server fornirà dei consigli in base alle parole provate. In particolare il consiglio sarà formato dalla stessa parola che è stata utilizzata come guess con le lettere colorate secondo il seguente schema:
  - Per ogni lettera della parola se:
    - la lettera è verde se è presente nella parola ed è nella posizione giusta;
    - la lettera è gialla se è presente nella parola ma non è nella posizione giusta;
    - la lettera è bianca se non è presente nella parola da indovinare.

  Una volta terminato il gioco sarà stampato il risultato sullo schermo e sarà chiesto se si vuole condividere i proprio risultati eseguendo il comando "**share**" o uscire eseguendo il comando "**exit**".

- **showMeSharing**: Rende possibile vedere i risultati che sono stati condivisi dagli altri giocatori. I risultati che saranno visibili sono solo quelli che sono stati condivisi dopo il proprio primo login.

Ogni volta finita l'esecuzione di uno dei comandi viene richiesto quale si vuole eseguire consecutivamente.

Da notare che tutti i comandi eccetto play una volta eseguiti devono essere portati a completamento. Nel caso di play è possibile “interrompere” l’esecuzione a metà facendo il logout.

L’utente termina l’esecuzione del programma chiudendo la console da cui è stato eseguito (e.g. chiudendo il cmd su Windows o la bash console su Linux)