Università di Pisa

Dipartimento di Informatica

Relazione del progetto `Wordle`

Autore: Riccardo Mori
Matricola: 505163
Marzo 2023

Indice

1 Descrizione del progetto

Il progetto consiste nell'implementazione (in Java 8) di una versione semplificata del famoso gioco **Wordle**. In particolare questa variante del gioco supporta numero di tentativi e lunghezza delle parole variabili[1].
Il progetto si compone di due parti indipendenti fra loro: il **server** e il **client**.

La struttura e' la seguente:

```
/
├── client/   Implementazione del client
│   ├── backend/            Offre un'implementazione di base
│   │                       delle funzionalita' supportate dal
│   │                       client
│   │   └── exceptions/
│   └── frontend/           Implementazione frontend
│       ├── CLI/   Command Line Interface
│       └── GUI/   Graphical User Interface (Opzionale)
├── protocol/               Costanti e strutture dati
│                           condivise relative al protocollo di
│                           comunicazione client-server over TCP
├── rmi/   Interfacce e strutture dati condivise
│   │      relative a RMI
│   └── exceptions/
├── server/                 Implementazione del server
│   └── logging/
└── utils/   Utilities
```

2 Protocollo di comunicazione

Il protocollo di comunicazione client-server e' stato implementato avendo come obiettivi:

- La dimensione dei messaggi inviati
- La facilita' del parsing dei messaggi
- L'indipendenza del protocollo da qualsiasi implementazione, linguaggio di programmazione, sistema operativo o architettura utilizzata

E' stato dunque scelto un protocollo custom binario in cui i messaggi vengono codificati in **UTF-8** e incapsulati all'interno di un pacchetto (come si puo' vedere in Figura 1) nel seguente formato:

1. Dimensione in byte del messaggio codificato in 4 bytes BIG_ENDIAN
2. Il messaggio vero e proprio (codificato in $UTF\text{-}8$)

[1] I valori sono impostati staticamente (*hardcoded*)

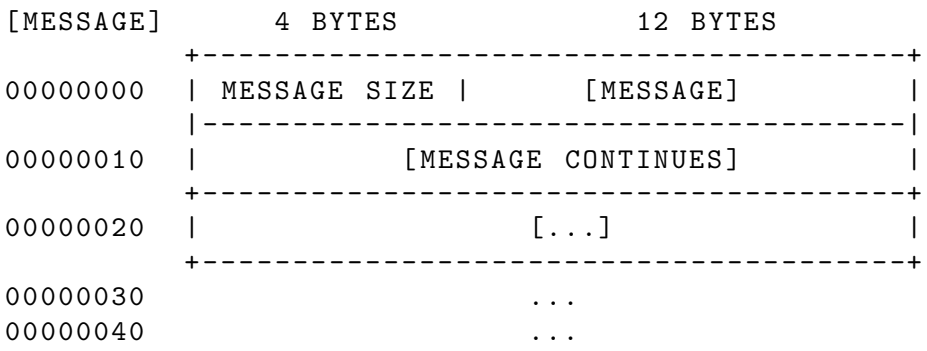

Figura 1: Rappresentazione dell'incapsulamento di un messaggio.

Per una descrizione approfondita del formato dei messaggi consultare l'appendice A

3 Server

3.1 Architettura

La struttura del server (raffigurata in figura 2) e' la seguente:

```
server/
├── ServerMain   Entry point principale
├── WordleServer   Core del server, istanza singleton
├── ClientSession   Gestisce la sessione di un client
├── Leaderboard   Struttura dati che contiene la
│                 classifica degli utenti
├── TranslationServer   Offre la traduzione in italiano delle
│                       secret word
├── User   Rappresenta un utente
├── UserSession   La sessione di un utente
└── logging/
```

Il nucleo principale del server e' contenuto nella classe **WordleServer** che e' anche un oggetto singleton poiche' deve essere istanziato la prima volta da **ServerMain** e da quel momento in poi non deve essere mai piu' istanziato nuovamente, ne' terminato.

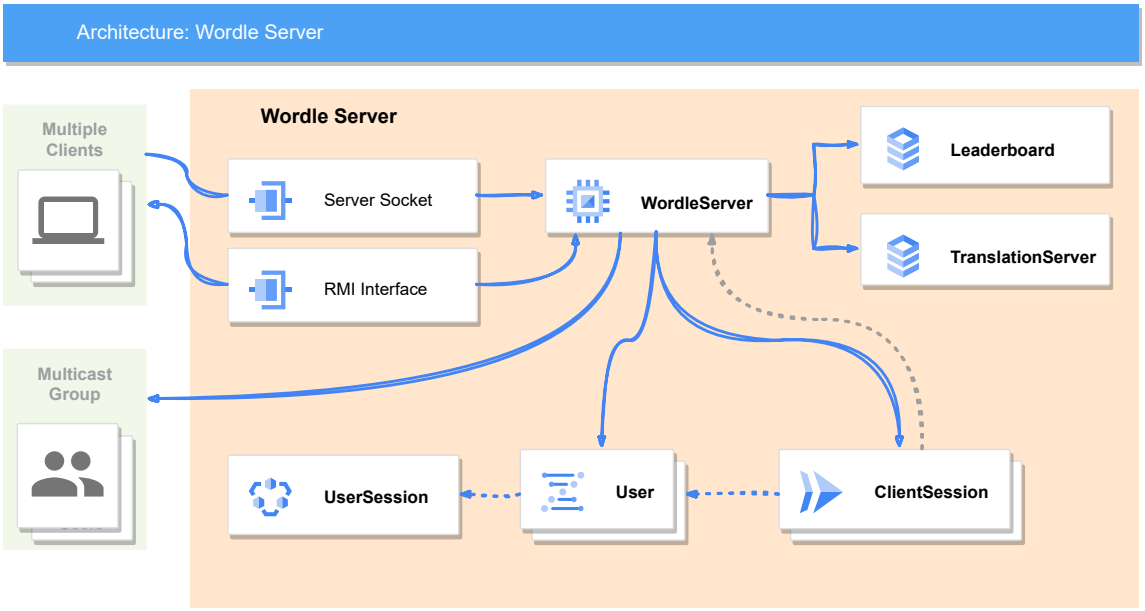

Figura 2: Rappresentazione dell'architettura del server. La linea tratteggiata blu indica che il riferimento e' opzionale (potrebbe non essere presente). Si noti che esiste una dipendenza circolare tra **WordleServer** e **ClientSession**, sarebbe stato possibile evitarla rendendo **ClientSession** una classe `private static` di **WordleServer** ma si avrebbe perso in chiarezza.

WordleServer si occupa di:

- Gestione della comunicazione con i client (tramite TCP, UDP e java RMI)
- Mantenimento delle informazioni persistenti relative agli utenti
- Registrazione e autenticazione degli utenti
- Generazione e validazione delle parole segrete
- Condivisione delle partite nel gruppo di multicast
- Traduzione della *secret word*
- Notificare i client (*subscribers*) quando avviene un cambiamento nelle prime posizioni in classifica

Tuttavia **WordleServer** non e' direttamente responsabile della logica della interazione con i client che invece e' affidata a **ClientSession**, il quale e' stato implementato come un semplice automa a stati finiti (raffigurato in Figura 3), cosa che permette di verificarne facilmente la correttezza e quindi di garantire che il client si trovera' sempre in uno stato *"safe"*.

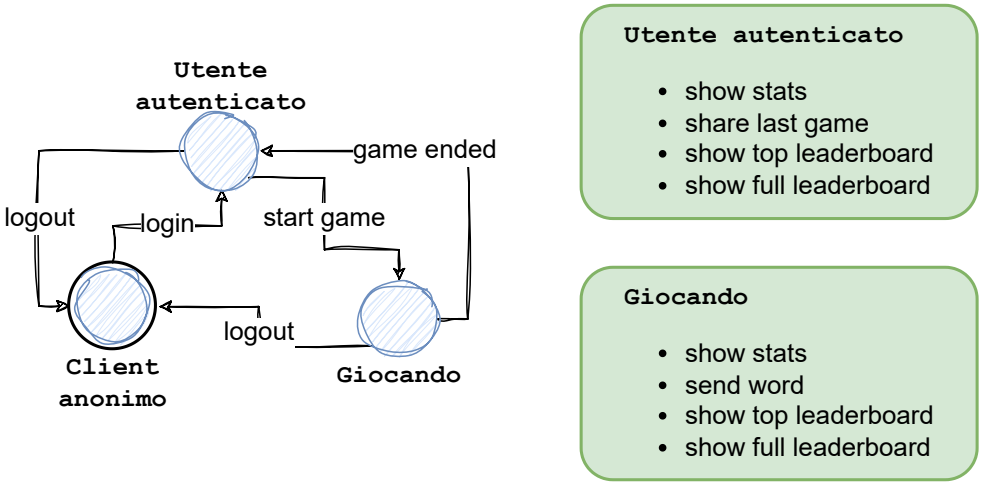

Figura 3: A sinistra: descrizione dell'automa a stati finiti che gestisce l'interazione di un client col server (lo stato iniziale e' quello cerchiato). A destra: elenco dei comandi validi in ogni stato (questi comandi non cambiano lo stato corrente dell'automa).

3.2 TCP Socket

Si e' scelto di gestire le connessioni TCP con dei socket I/O non bloccanti (*Java NIO*) piuttosto che avere un pool di thread in cui ogni thread e' responsabile di un solo client.

Questa soluzione ci permette una migliore scalabilita' col numero di client, infatti un elevato numero di client che effettuano tuttavia poche operazioni causerebbe un enorme overhead se usassimo un thread ciascuno. In questo modo siamo limitati da un solo thread che gestisce le connessioni con dei dati pronti a essere letti o scritti.

In realta' la limitazione del thread singolo potrebbe essere rimossa qualora si sentisse il bisogno[1] di migliorare ulteriormente l'efficienza sfruttando il parallelismo hardware (piu' core/CPU). In tal caso basterebbe delegare l'esecuzione di ciascun **ClientSession** a un pool di thread introducendo delle misure di sincronizzazione per evitare di elaborare due volte lo stesso messaggio.

3.3 Leaderboard

La classifica degli utenti e' gestita dalla classe **Leaderboard**. Internamente la classifica viene codificata con due strutture dati:

- Un **BST** (*Binary Search Tree*) in cui i nodi (che rappresentano gli utenti)

[1] Questa soluzione non e' stata adottata poiche' un tale livello di efficienza esula dagli obiettivi di questo progetto e l'implementazione risulta particolarmente insidiosa, tuttavia il resto del codice e' gia' stato predisposto per supportare una gestione parallela dei client

sono ordinati per il **punteggio** dell'utente e sono etichettati con il relativo **username** (che ricordiamo essere unico)

- Una **hashmap** in cui la chiave e' lo **username** (se e solo se e' presente in classifica) e il valore associato e' un riferimento al nodo nel **BST** che identifica proprio quello username.

In questo modo sapere la posizione in classifica di un utente richiede $O(n)$ tempo mentre l'aggiunta, la rimozione e la modifica di una coppia (utente, punteggio) richiede $O(\log n)$ tempo.

Vale la pena notare che usare un **Order Statistic Tree** al posto del **BST** comporterebbe un miglioramento della complessita' dell'operazione di *rank*, ovvero sapere la posizione in classifica di un utente, che diventerebbe quindi $O(\log n)$. Siccome l'implementazione di un **OST** non e' presente nella standard library di Java questa soluzione non e' stata utilizzata.

Praticamente l'implementazione non rispecchia totalmente la struttura definita sopra ma comporta alcune piccole modifiche che non cambiano la complessita' del problema:

- Il **BST** e' stato implementato mediante una **TreeMap** in cui la chiave e' la coppia (ordinabile lessicograficamente) *(punteggio, username)* mentre il valore non e' utilizzato.
- La **hashmap** e' stata implementata tramite una **HashMap** la cui chiave e' uno *username* e il valore e' la coppia *(punteggio, username)*.

Per mantenere sincronizzate entrambe le strutture dati e' necessario che tutti i metodi della classe **Leaderboard** siano mutualmente esclusivi, per fare cio' e' sufficiente dichiararli **synchronized**.

3.4 Traduzione della secret word

La traduzione della *secret word* (gestita dalla classe **TranslationServer**) in italiano e' una procedura lenta che richiede di eseguire una richiesta HTTP e siccome si e' deciso di non utilizzare un thread pool per la gestione concorrente dei client l'intero server si blocca in attesa della risposta dal web server.

Per mitigare questo problema internamente viene usata una cache **LRU** (*Least Recently Used*) che contiene la traduzione delle secret word. L'implementazione della cache **LRU** e' basata sulla **LinkedHashMap** offerta dalla standard library.

3.5 Gestione della concorrenza

Come precedentemente anticipato non ci sono molti thread che lavorano in parallelo. Tuttavia alcune componenti sono eseguite in un thread separato, in particolare la lista dei thread e':

- Il main thread

- Flush scheduler: viene eseguito a intervalli regolari ed esegue la sincronizzazione con il file di salvataggio del server
- Thread per la generazione di una nuova secret word. Anche questo viene eseguito a intervalli regolari
- Shutdown hook: thread speciale che viene eseguito nel momento in cui il server inizia la fase di terminazione. Esegue un sync sul file di salvataggio.
- Share game: il thread si occupa di condividere una partita nel gruppo di multicast. Essendo questa di per se' una attivita' asincrona puo' essere eseguita tranquillamente da un thread separato
- Notifica ai subscribers. Anche questa attivita' puo' essere eseguita parallelamente siccome non viene generata direttamente da un client
- RMI threads: I metodi remoti disponibili possono generare molteplici thread. Il threadpool in questo caso e' gestito dalla *JVM*

E' quindi necessario fare in modo che non si presentino situazioni di deadlock ne' altri problemi.

La maggior parte di questi thread opera su strutture dati dedicate o su tipi di dato primitivi (l'assegnamento in tal caso e' atomico), quindi le misure per la prevenzione di problemi dovuti all'accesso concorrente alle risorse sono molto semplici o addirittura assenti.

L'unica struttura dati che merita particolare attenzione e' la **map** $\{username : User\}$ che associa a ogni *username* il relativo oggetto di tipo *User* che rappresenta l'utente. Tale struttura e' particolarmente delicata per i seguenti motivi:

1. In fase di registrazione bisogna garantire l'atomicita' dell'operazione di creazione di un utente se gia' non esiste
2. L'autenticazione, che consiste nel controllare se un utente esiste e in tal caso controllare se la password e' valida, deve essere atomica
3. Durante la sincronizzazione dello stato del server in un file di salvataggio e' necessario operare su uno snapshot della lista di utenti **evitando** di scrivere informazioni corrotte o parziali
4. Ogni istanza di **ClientSession** puo' operare sul relativo oggetto **User** ma non opera direttamente sulla **map** $\{username : User\}$

Per risolvere questi problemi e mantenere comunque un alto livello di parallelismo si e' deciso di utilizzare una **ConcurrentHashMap** che, unita alle proprieta' che gli utenti non si possono eliminare e la password non si puo' cambiare, ci permette di risolvere i punti 1-2.

Per quanto riguarda il punto 4 si e' deciso di imporre un'altra proprieta':

> In ogni istante successivo alla inizializzazione del server solo un oggetto di tipo **ClientSession** puo' operare su un determinato oggetto **User**

Infine il punto 3 si risolve imponendo che la classe **User** sia serializzabile previa cattura del lock tramite **synchronized**. Questo significa che le operazioni di modifica su **User** sono tutte atomiche e lasciano l'oggetto in uno stato corretto.

3.6 Configurazione

Nel file di configurazione `ServerMain.properties` si possono configurare i seguenti valori:

- `multicast_address`: l'indirizzo del gruppo di multicast
- `multicast_port`: la porta da utilizzare quando si mandano messaggi nel gruppo di multicast
- `server_port`: la porta da usare per il server socket
- `rmi_port`: la porta da usare per i servizi RMI
- `verbose`: la verbosita' dei log (da 0 a 5)
- `secret_word_rate`: l'intervallo in secondi tra la generazione di una secret word e la successiva
- `words_db`: file contenente la lista di parole

4 Client

Il client e' diviso in due parti: **backend** e **frontend**. Il primo implementa tutte le varie funzioni che prevedono un'interazione con il server in modo da offrire una sorta di *API* comune a qualsiasi frontend, il secondo invece e' l'interfaccia utente vera e propria.

Questa scelta permette di riutilizzare il codice e separare ulteriormente l'interfaccia dalle funzionalita'.

4.1 Backend

La struttura del backend e' la seguente:

```
client/backend/
├── ClientBackend   Implementazione delle funzionalita'
│                   del client
├── NotificationListener   Riceve i messaggi da un gruppo di
│                          multicast
└── exceptions/   Eccezioni custom
```

La classe principale e' ovviamente **ClientBackend** che implementa la quasi totalita' delle funzionalita' richieste dal client (login, inizio di una partita, tentativo di indovinare la parola segreta, ...). E' importante notare che le funzionalita' offerte non controllano la consistenza dello stato del sistema ma sono una

vera e propria *API*. Sara' compito del frontend effettuare le chiamate giuste al momento giusto.

I metodi offerti da **ClientBackend** si occupano di creare il messaggio da inviare al server (tramite socket TCP) e in caso di successo terminano ritornando gli eventuali dati ricevuti dal server (corretamente parsati), altrimenti sollevano un'eccezione che estende **BackendException** indicativa del problema avvenuto. Nel caso in cui il messaggio di risposta dal server contenga dei dati aggiuntivi, nonostante l'operazione richiesta non sia andata a buon fine, tali dati, opportunamente parsati, verranno incapsulati all'interno dell'eccezione sollevata e potranno essere recuperati tramite il metodo **BackendException::getResult**. Se ne puo' vedere un esempio in **AlreadyPlayedException**.

Questo sistema permette una maggiore flessibilita' nella comunicazione tra **frontend** e **backend**, infatti l'utilizzatore del **backend** sapra' sempre qual e' il tipo di dato ritornato e quali le eccezioni generate. Alternativamente, ritornare sempre un tipo di dato che contemporaneamente codifichi (tipicamente tramite un intero) sia il successo che tutti i possibili errori e al tempo stesso anche tutti i possibili dati aggiuntivi comunicati dal server, avrebbe reso il codice meno leggibile, piu' prono a errori e meno auto-esplicativo.

Un'ultima considerazione va fatta per **NotificationListener** che lancia un thread nel quale resta in ascolto di eventuali messaggi su un gruppo multicast. Gli unici problemi causati dalla concorrenza in questo caso sono dovuti agli accessi alle risorse interne alla classe, quindi sono molto semplici da gestire.

4.2 Frontend

Il frontend e' offerto da due diverse implementazioni: una **CLI** (*Command Line Interface*) e una **GUI** (*Graphical User Interface*).

La **GUI**, essendo facoltativa, non verra' discussa nel dettaglio. Viene offerta *as is* semplicemente per completezza.

La struttura del frontend e' la seguente:

```
client/frontend/
├── CLI/
│   └── ClientCLI
├── ClientFrontend   Interfaccia che ogni frontend deve
│                    implementare
├── Command
├── SessionState   Mantiene lo stato del client
└── GUI/
```

Essendo che il client deve implementare l'interfaccia **clientRMI** in cui vengono specificati i metodi remoti che possono essere invocati dal server (che rappresenta una linea diretta di comunicazione col server), e' chiaro che il frontend dovra' implementare tali metodi, senza poter demandare al backend questa responsabilita'.

Fortunatamente c'e' solo un metodo (**updateLeaderboard**) invocabile da remoto ed e' di facile implementazione. Cio' non causa problemi di concorrenza.

Il resto del client e' composto da un singolo thread (il principale). Questo, unito al fatto che si appoggia al backend per l'implementazione delle funzionalita' richieste, rende la classe **ClientCLI** semplice e lineare.

4.3 Configurazione

Nel file di configurazione `ClientMain.properties` si possono configurare i seguenti valori:

- `multicast_address`: l'indirizzo del gruppo di multicast
- `multicast_port`: la porta da utilizzare quando si mandano messaggi nel gruppo di multicast
- `server_host`: l'host del server
- `server_port`: la porta da usare per connettersi al server
- `rmi_port`: la porta da usare per i servizi RMI

5 Manuale d'uso

Per la compilazione e l'esecuzione e' richiesto **Java 8** o superiore.

Esistono diversi modi per compilare il progetto:

- **Makefile** (Linux, MacOS, Unix-like)
- **Gradle** (All platforms)
- **Manualmente** (All platforms)

5.1 Makefile

Per compilare

```
$ make all
```

Per creare i files `jar`

```
$ make jar
```

Il risultato della compilazione sara' nella cartella `build/`

5.2 Gradle

5.2.1 Unix

Di seguito verra' descritta la procedura per sistemi Unix-like (Linux, MacOS, ...)

Per compilare

```
$ ./gradlew build
```

Il risultato sara' nella cartella `app/build/`

Per creare i files `jar`

```
$ ./gradlew clientJar
$ ./gradlew serverJar
```

Il risultato sara' nella cartella `app/build/libs/`

5.2.2 Windows

Di seguito verra' descritta la procedura per sistemi Windows

Per compilare

```
gradlew.bat build
```

Il risultato sara' nella cartella `app\build\`

Per creare i files `jar`

```
gradlew.bat clientJar
gradlew.bat serverJar
```

Il risultato sara' nella cartella `app\build\libs\`

5.3 Metodo manuale

Non raccomandato

5.3.1 Unix

Di seguito verra' descritta la procedura per sistemi Unix-like (Linux, MacOS, ...)

Per compilare

```
$ javac -cp lib/gson-2.10.1.jar:./app/src/main/java: -d ./build \
    app/src/main/java/edu/riccardomori/wordle/server/ServerMain.java
$ javac -cp lib/gson-2.10.1.jar:./app/src/main/java: -d ./build \
    app/src/main/java/edu/riccardomori/wordle/client/ClientMain.java
```

Il risultato sara' nella cartella `app/build/`

5.3.2 Windows

Di seguito verra' descritta la procedura per sistemi Windows

Per compilare

```
javac.exe -cp lib\gson-2.10.1.jar;.\app\src\main\java; -d .\build \
app\src\main\java\edu\riccardomori\wordle\server\ServerMain.java
javac.exe -cp lib\gson-2.10.1.jar:.\app\src\main\java: -d .\build \
app\src\main\java\edu\riccardomori\wordle\client\ClientMain.java
```

Il risultato sara' nella cartella `app\build\`

5.4 Esecuzione

Per eseguire l'applicazione e' consigliato usare i file `jar` precedentemente compilati nel seguente modo:

```
$ java -jar client.jar
$ java -jar server.jar
```

Altrimenti e' possibile lanciare l'applicazione manualmente nel seguente modo:

Unix:

```
$ java -cp lib/gson-2.10.1.jar:./build/ \
     edu.riccardomori.wordle.client.ClientMain
$ java -cp lib/gson-2.10.1.jar:./build/ \
     edu.riccardomori.wordle.server.ServerMain
```

Windows:

```
java.exe -cp lib\gson-2.10.1.jar;.\build edu.riccardomori.wordle.client.ClientMain
java.exe -cp lib\gson-2.10.1.jar;.\build edu.riccardomori.wordle.server.ServerMain
```

6 Note

Il codice in questione si puo' trovare all'indirizzo `https://github.com/patacca/wordle-Unipi`

La versione qui descritta e' quella identificata dal commit xxx

A Formato dei messaggi