Relazione progetto di reti *Wordle*

Leonardo Scoppitto

Marzo 2023

Indice

1 Utilizzo e parametri di configurazione

1.1 Compilazione ed esecuzione

Per realizzare e testare il progetto è stata utilizzata la versione 16 di OpenJDK in ambiente Linux. Una volta posizionati nella root del progetto, i comandi necessari alla compilazione sono:

```
find . -name "*.java" > ./files.txt
javac -d "./build" -cp "./lib/jackson-core-2.13.0.jar:./lib/jackson-annotations-2.13.0.jar:./lib/jackson-databind-2.13.0.jar:./lib/sqlite-jdbc-3.40.0.0.jar" @files.txt
```

Per eseguire il progetto appena compilato, basterà dare i comandi:

```
# Client
java -cp "./lib/jackson-annotations-2.13.3.jar:./lib/jackson-core-2.13.3.jar:./lib/jackson-databind-2.13.3.jar:./build/" Client.ClientMain </path/to/config.json>

# Server
java -cp "./lib/jackson-annotations-2.13.3.jar:./lib/jackson-core-2.13.3.jar:./lib/jackson-databind-2.13.3.jar:./lib/sqlite-jdbc-3.40.0.0.jar:./build/" Server.ServerMain </path/to/config.json>
```

In alternativa è possibile spostarsi nella root del progetto, eseguire lo script si compilazione `compileWordle.sh` per poi eseguire `execServer.sh` e `execClient.sh` passando come argomento il path dei file di configurazione. Potrebbe essere necessario aggiungere i permessi di esecuzione con il comando `chmod +x *.sh` Per quanto riguarda la configurazione degli applicativi ho deciso di utilizzare il formato JSON de-serializzato nelle classi `ServerConfig` e `ClientConfig`, entrambe implementate come Record.

1.1.1 Problemi noti

Durante la fase di test del progetto ho riscontrato i seguenti problemi:

- Per far funzionare la comunicazione attraverso il gruppo multicast quando si esegue il client e il server sulla stessa macchina, è stato necessario, almeno nel mio caso su Fedora 37, disabilitare momentaneamente il firewall[1]. Se il client e il server sono in esecuzione su due computer diversi, invece, i pacchetti non vengono ricevuti dai client connessi.
- Lo stub esportato dal client per registrare la callback RMI sul server potrebbe bloccare l'esecuzione del client se si testa il progetto eseguendo il server e il/i client su pc diversi . Il client, se ha più interfacce di rete, non è detto che sceglierà quella corretta, infatti nel mio caso testando il progetto connesso alla rete wifi (interfaccia n. 2, `wlp59s0`), non avevo problemi, mentre adoperando un adattatore per l'utilizzo del cavo lan (interfaccia n. 46 `enp0s20f0u2u3c2`), il client si fermava sulla chiamata `remoteRankingObject.addListener(rankingListenerStub)` e tramite il debugger ho visto che l'indirizzo ip usato era quello relativo a un'interfaccia di rete virtuale (interfaccia n. 8, 'br-05695febda49').

1.2 Configurazione del Server

Di seguito un esempio di configurazione del server:

```
{
  "tcpPort" : 6789,
  "rmiPort" : 3800,
  "multicastPort" : 7800,
  "multicastAddress" : "224.0.0.1",
  "corePoolSize" : 4, // corePoolSize del cachedTreadPool che gestirà le richieste
  "secretWordTimeout" : 120000, // Intervallo in millisecondi fra l'estrazione di due SW
  "verbose" : false, // Se true, stampa a schermo ogni richiesta in arrivo dal client e relativa risposta
  "sqliteDatabasePath" : "/home/leonardo/Documents/University/Reti/Laboratorio/Wordle_Project/Wordle/src/testDb-3.db",
  "autoCreateDatabase" : true, // Se true, crea i file necessari per il funzionamento del DB
  "autoCommitDatabase" : false, // Se true viene attivato l'auto-commit del database, vedere sezione 2.4
  "wordsFilePath" : "/home/leonardo/Documents/University/Reti/Laboratorio/Wordle_Project/Wordle/src/words.txt",
  "useJsonDatabase" : true,
  "jsonDatabasePath" : "/home/leonardo/Documents/University/Reti/Laboratorio/Wordle_Project/Wordle/src/JsonDatabase",
  "debug" : "true" // Se true, viene abilitato l'endpoint getCurrentWord
}
```

Anche se può sembrare ridondante, ho deciso di mantenere due voci separate per i path dei database in modo da poter testare più agevolmente entrambe le soluzioni senza dover modificare la configurazione ogni volta che si va a cambiare il flag `useJsonDatabase`.

N.B.: Tutti i campi sono necessari affinché il server vada in esecuzione.

[1] Si potrebbero creare anche nuove regole, ma per un semplice test ho preferito fare così

1.3 Configurazione del Client

Di seguito un esempio di configurazione del client:

```
{
  "tcpPort" : 6789,
  "rmiPort" : 3800,
  "maxRetries" : 5, // Numero massimo di tentativi di riconnessione
  "reconnectionTimeout" : 1500, // Timeout fra un tentativo e l'altro
  "networkInterface" : "wlp59s0", // Interfaccia su cui ricevere i DatagramPackets
  "serverIp" : "localhost",
  "rmiRegistrationEndpoint" : "REGISTRATION",
  "rmiRankingEndpoint" : "RANKING",
  "language" : "it-it",
  "prettyPrint" : true // Se true, saranno abilitati i colori nella CLI
}
```

Le lingue supportate sono l'italiano (`it-it`) e l'inglese (`en-en`).

N.B.: Tutti i campi sono necessari affinché il client vada in esecuzione.

2 Server

2.1 Protocolli usati per la comunicazione

L'applicazione server espone un'API accessibile mediante richieste HTTP tramite la quale, come vedremo più avanti, è possibile interagire sia tramite il client CLI, che tramite un browser. Inoltre, il client CLI, per particolari operazioni, interagisce col server tramite RMI e multicast socket, come da specifica. Qui una lista di tutti gli endpoint:

Endpoint	Protocollo	Parametri	Descrizione
REGISTRATION	RMI	username, password	Registrazione tramite client CLI
RANKING	RMI	rankingListener	Registrazione dell'RMI callback
/register	HTTP POST	username, password	Registrazione tramite API HTTP (realizzata per permettere la registrazione tramite webapp)
/login	HTTP POST	username, password	Login tramite CLI/webapp. Il client riceve dal server il token per autenticare le richieste
/verify	HTTP GET	username, token	Verifica del token salvato nel *local-storage* del browser
/logout	HTTP POST	username, token	Logout tramite CLI/webapp
/playWordle	HTTP POST	username, token	`username` crea una nuova partita e riceve il wordId per iniziare a giocare
/sendWord	HTTP POST	username, token, word, wordId	Invia una parola al server per tentare di indovinare la SW
/showMeRanking	HTTP GET	username, token	Restituisce la classifica (Top 10) aggiornata e ordinata dalla prima all'ultima posizione
/sendMeStatistics	HTTP GET	username, password	Restituisce le statistiche aggiornate dell'utente
/share	HTTP POST	username, password, wordId	Invia una notifica agli altri utenti facenti parte del gruppo multicast
/getMulticast	HTTP GET	username, token	Restituisce l'indirizzo e la porta dove avverranno le comunicazioni del gruppo multicast
/getGameStatus	HTTP GET	username, token	Restituisce `isPlaying` e `wordId` associati all'utente `username`
/getGameHistory	HTTP GET	username, token, wordId	Restituisce tutti i tentativi dell'utente e i relativi suggerimenti associati alla SW `wordId`
/wordTimer	HTTP GET	username, token	Restituisce `time`, il tempo in millisecondi rimanente per indovinare la parola
/getCurrentWord	HTTP GET		Quando il flag `debug` vale `true`, restituisce la SW, altrimenti restituisce `Method not allowed`

2.2 Autenticazione

Come metodo di autenticazione è stato scelto il protocollo *bearer token*, implementato inserendo l'header `Authorization` alle richieste o, in alternativa, passando il token come parametro qualora non si potesse modificare l'header della richiesta. Il token, ricevuto dal client in risposta a una richiesta di login, consiste in una stringa di 32 byte che non sono altro che la rappresentazione esadecimale del *message digest* prodotto applicando l'algoritmo `SHA-256` alla stringa ottenuta componendo il timestamp dell'ora corrente, l'username dell'utente che esegue il login e un intero casuale generato mediante la classe `SecureRandom`. Dell'hash prodotto, lungo originariamente 256 bit, viene poi presa una porzione random lunga 128 bit.

```
private String generateSessionToken(String username) {
        SecureRandom random = new SecureRandom();
        MessageDigest digest = null;
        byte[] hash;
        String hashString;
        String time = String.valueOf(System.currentTimeMillis());
        String rand = String.valueOf((int) (random.nextDouble() * 100000));
        String toHash = time + username + rand;
        int low, high;
        try {
            digest = MessageDigest.getInstance("SHA-256");
        } catch (NoSuchAlgorithmException ignored) {}
        if (digest != null){
            hash = digest.digest(toHash.getBytes(StandardCharsets.UTF_8));
            StringBuilder hexString = new StringBuilder(2 * hash.length);
            for (byte b : hash) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) {
                    hexString.append('0');
                }
                hexString.append(hex);
            }
            hashString = hexString.toString();
            low = random.nextInt((hashString.length() - 1) / 2);
            high = low + (hashString.length() / 2);

            return hashString.substring(low, high);
        }
        return "";
    }
```

Il token così generato, viene quindi memorizzato dal client in un'istanza della classe `Session`, in modo che questi possa utilizzarlo fino al logout o fino alla scadenza del token[2], mentre nel server la classe `LoginHandler` si occupa di gestire le sessioni attive mediante una `ConcurrentHashMap` avente per chiave l'username dell'utente e come valore un'istanza della classe `TokenSession`, che contiene il token, il timestamp dell'ultimo login e il tipo di login. Per lasciare spazio a una futura implementazione di metodi per l'amministrazione del database e del server, è previsto un campo per salvare il ruolo dell'utente, al momento limitato a `user` o `admin`.
Il salvataggio delle password nel database non avviene in chiaro, bensì viene calcolato l'hash (algoritmo `SHA-256`) della concatenazione della password con un numero random generato tramite la classe `SecureRandom` (il cosiddetto *salt*) e poi viene inserita nel database la stringa `salt:hashedPassowrd`. Al momento del login, il sistema recupera il salt, lo unisce alla password ricevuta dall'utente e confronta l'hars ottenuto con quello salvato nel database. In questo modo anche se due utenti non hanno la stessa password, avranno due hash completamente diversi. Di seguito le due porzioni di codice che gestiscono il salvataggio e la verifica della password:

```
// registrazione
salt = String.valueOf((int) (random.nextDouble()*100000));
digest = MessageDigest.getInstance("SHA-256");
hashedPassword = salt + ":" + bytesToHex(digest.digest((salt + password).getBytes(StandardCharsets.UTF_

// login
salt = hashedPasswd.split(":")[0];
hashedPasswd = hashedPasswd.split(":")[1];
if(hashedPasswd.equals(bytesToHex(digest.digest((salt + password).getBytes(StandardCharsets.UTF_8)))))
    // utente autenticato
}
```

[2]Non è prevista una scadenza vera e propria, nonostante si faccia un rinnovo della sessione ad ogni login, pertanto al momento la scadenza coincide col riavvio del server in quanto i token non vengono salvati sulla memoria permanente

2.3 Gestione delle richieste

Per la gestione delle connessioni è stato scelto il multiplexing dei canali tramite `Selector NIO`, con successiva presa in carico della richiesta da parte di un `cachedThreadPool`. Il server, dopo aver letto il file di configurazione, aver esportato gli stub necessari al funzionamento dei servizi RMI, aver avviato il database e infine il servizio di estrazione delle parole, apre un `ServerSocketChannel` che verrà registrato su un `Selector` per iniziare ad accettare connessioni da parte dei client. Il thread main del processo server rimarrà per tutta l'esecuzione del programma in un ciclo dove accetterà le richieste di connessione per poi registrare le nuove chiavi sul selector. Quando il metodo `selector.select()` restituisce invece delle chiavi marcate come *readable*, verrà letta la richiesta del client, la quale verrà passata a una nuova istanza della classe *runnable* `RequestHandler`, mandata poi in esecuzione sul thread pool. Una volta completate le operazioni richieste, l'esito verrà scritto all'interno di una `Hashmap<String, Object>`, serializzata poi in JSON e scritta quindi su un `ByteBuffer`, in modo che possa essere allegata alla chiave tramite il metodo `attachJson`. Alle chiavi *writeable*, il thread main invierà quindi l'*attachment* contenente la risposta tramite il metodo statico `RequestHandler.sendAttachment()`.

È stato scelto questo approccio essendo specificamente richiesto di mantenere attiva la connessione col client per tutta la durata della sessione: l'utilizzo di librerie di `IO` tradizionale avrebbe richiesto di mantenere un thread attivo per ogni client connesso, a scapito dell'uso efficiente delle risorse. Registrando i canali sul selector, invece, i thread worker verranno attivati solo quando vi è effettivamente del lavoro da fare, riducendo al minimo la quantità di thread usati dal programma. Infine, è stato scelto l'utilizzo di un `cachedThreadPool` così da poter avere un certo numero di thread (configurabile dall'utente) sempre pronti ad eseguire le richieste, ma avere anche, all'occorrenza, la possibilità di aumentare dinamicamente il numero di thread attivi nei momenti di picco del numero delle richieste.

2.3.1 La classe `RequestHandler`

Quando il thread main riceve una nuova richiesta, viene creata un'istanza del task `RequestHandler` che implementa l'interfaccia `Runnable`, potendo così essere mandato in esecuzione sul thread pool. La richiesta HTTP, ricevuta sotto forma di stringa, viene letta riga per riga tramite un `BufferedReader` estraendo così l'*header* e il *body*, che verranno passati al metodo `handleRequest`: come prima cosa vengono estratti i parametri dalla *query string*, dal *body* e dall'*header*, per poi venire validati insieme all'endpoint e al tipo della richiesta nel metodo `getEndpoint`. Questo restituisce un valore dell'enumerazione `Endpoint` che verrà usato all'interno di `handleRequest` dallo switch statement per eseguire le operazioni volute e popolare di conseguenza l'`HashMap` contenente la riposta. Quest'ultima verrà poi serializzata dal metodo `attachJson`, che riceve in input anche lo *status code* della risposta HTTP, e allegata alla chiave tramite il metodo `attach`: a questo punto la chiave viene marcata come *writeable* e viene invocato il metodo `selector.wakeup()` per svegliare il thread main e farlo procedere alla scrittura della risposta.

2.4 Database e gestione dei dati

Dell'interazione con la memoria permanente e della persistenza dei dati se ne occupa la classe `WordleDB`, che a sua volta si appoggia alle classi `WordleSqliteDB` e `WordleJsonDB` le quali contengono l'implementazione vera e propria dei metodi che permettono il salvataggio, la manipolazione e il recupero delle informazioni riguardanti gli utenti registrati, le sessioni di gioco e le *secret words* utilizzate dal server. In questo modo è più facile mantenere e aggiornare il codice che si occupa della logica dietro alle operazioni sul database, permettendoci, ad esempio, di poter aggiungere un altro tipo di database (NoSQL, a oggetti, ecc.) senza dover cambiare il codice dalla parte del chiamante. Il server all'avvio legge il file di configurazione e controlla quale tipo di database si vuole usare tramite il flag `useJsonDatabase` così da usare il path corretto fra `sqliteDatabasePath` e `jsonDatabasePath`, in corrispondenza del quale si trovano i file associati al database[3].

2.4.1 Implementazione di `WordleJsonDB`

La classe `WordleJsonDB` contiene l'implementazione del database tramite più mappe di tipo `ConcurrentHashMap` e successiva serializzazione e de-serializzazione su file in formato JSON. I file JSON (`users.json`, `games.json` e `words.json`) rappresentano le tabelle del database SQLite così da mantenere una consistenza logica fra i due. È stata scelta come struttura dati la `ConcurrentHashMap` in modo da garantire ai thread che gestiscono le richieste la possibilità di accedevi in modo concorrente, mantenendo un buon livello di performance rispetto ad altre strutture dati sincronizzate sia per quanto riguarda l'accesso concorrente, che per quanto riguarda le operazioni di inserimento e recupero delle informazioni. Sono state scelte come chiavi l'`username` per la tabella degli utenti, la coppia di valori `<username, word>` per la tabella *games* e `word` per la tabella contenente le parole estratte. In particolare, l'implementazione della coppia di valori, chiave della tabella *games*, è definita nella

[3]Se non esistono, verranno creati nel caso il flag `autoCreateDatabase` sia settato su `true`

classe `StringPair` dentro il *package* `CommonUtils`, mentre il metodo per de-serializzare gli oggetti di questo tipo si trova all'interno di `StringPairDeserializer`. All'inizio era stata implementata la classe `Pair`, che accetta come elementi della coppia due generici `T`, adattandosi anche a scenari in cui è necessario salvare due elementi di tipo diverso, ma dal momento che `username` e `word` sono entrambe stringhe e l'introduzione dei generici portava complicazioni nella serializzazione e de-serializzazione delle tabelle aventi come chiave un oggetto di questo tipo, ho deciso di creare una classe specifica. Le classi che modellano le *entries* del database sono definite nel package `Models`. Queste contengono le strutture dati per immagazzinare tutte le informazioni necessarie al funzionamento del server, in particolare, sono state usate strutture dati sincronizzate per le variabili che potenzialmente possono essere lette e scritte da più thread contemporaneamente.

2.4.2 Implementazione di `WordleSqliteDB`

Seppur non richiesto dalla specifica, ho deciso di implementare anche dei metodi per l'interazione con un database relazionale in modo da poter sperimentare l'uso di questa tecnologia in un contesto diverso dagli esercizi teorici di basi di dati.

La classe `WordleSqliteDB` contiene l'implementazione del database tramite la libreria SQLite. Dal sito ufficiale:

> *SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.*

Ho scelto questo tipo di database perché fonde la flessibilità e la potenza di un database relazionale con la semplicità di utilizzo, essendo *self-contained* e non necessitando di applicativi di terze parti o configurazioni particolari per funzionare, ma solo di un file su cui scrivere i dati, essendo per questo molto utilizzato in ambienti embedded/IoT e mobile. Per quanto riguarda l'implementazione, ho deciso di utilizzare solo metodi statici, in quanto non è necessario mantenere strutture dati per il corretto funzionamento del database, se non la connessione al database, che è però contenuta nell'istanza della classe `WordleDB` e viene passata come parametro ai metodi di `WordleSqliteDB`.
Inoltre, in questo caso la concorrenza è gestita internamente dal database stesso, pertanto non sono necessari particolari accorgimenti per la gestione dei thread.

2.4.3 Il flag `autoCommitDatabase`

A prescindere dal database in uso, è necessario fare una considerazione sulle performance. Vi sono principalmente due modi di gestire le scritture su disco dei dati contenuti in RAM:

- *auto-commit mode*: la scrittura (*commit*) dei dati viene eseguita dopo l'esecuzione di ogni query/operazione, garantendo così che la memoria persistente resti sempre aggiornata e limitando le perdite di dati in caso di crash del sistema, infatti si perderà al massimo il commit in esecuzione al momento del guasto.
- *manual-commit mode*: in questo caso viene eseguito il commit a intervalli di tempo regolari da una thread dedicato, facendo sì che a ogni operazione sul db, corrisponda solo un'operazione eseguita in RAM, quindi in maniera molto più rapida rispetto a prima dal momento che l'overhead dato dalla scrittura su disco viene diluito su più operazioni. In questo caso, però si ha una finestra temporale in cui i dati presenti in memoria temporanea non sono sincronizzati con quelli in memoria permanente e in cui si rischia un'ingente perdita di dati nel caso di crash del sistema.

Ho deciso di implementare entrambe le modalità, quindi nel caso si decidesse di disabilitare l'auto-commit, viene avviato un thread preposto alla gestione dei commit durante l'inizializzazione della classe `WordleDB`. Inoltre, lo *shutdown hook* installato all'avvio esegue un ultimo commit prima di terminare l'esecuzione del programma, assicurando, a meno di errori legati al database, che tutti i dati siano aggiornati al momento dell'uscita dall'applicazione.

2.5 La classe `WordFactory`

Questa classe implementando l'interfaccia `Runnable` viene eseguita in un thread dedicato e si occupa di estrarre una secret word sempre diversa a un intervallo specificato nella configurazione, oltre che a implementare il metodo che restituisce gli indizi a partire da una parola in input. Inoltre, ogni 5 secondi controlla se vi sono dei cambiamenti nella classifica, quindi li notifica ai client connessi attraverso il meccanismo di RMI callback. Qui la concorrenza viene gestita esplicitamente tramite `ReentrantReadWriteLock` così che i thread che gestiscono le richieste possano leggere contemporaneamente la secret word estratta o il timestamp senza doversi accodare dietro una lock normale.

2.6 Gestione degli errori e shutdown hook

Per cercare di preservare la consistenza dei dati, qualsiasi sia il database che si sta utilizzando, le eccezioni lanciate durante l'esecuzione del server vengono catturate e, nella maggior parte dei casi, il processo server termina chiamando `System.exit(1)`, così che non vengano eseguite operazioni che non possono essere salvate. Per le eccezioni legate alla rete, si è deciso di stampare semplicemente un errore, dal momento che un problema con un singolo client non deve andare a compromettere gli altri client. Allo spegnimento del server viene eseguita la routine di *cleanup* installata all'avvio tramite `Runtime.getRuntime().addShutdownHook()`: viene avviato un thread che invoca il metodo `shutdown()` sul thread pool, vengono chiusi i thread ausiliari, vengono cancellate tutte le chiavi registrate sul selector e chiuso il database. Dopodiché lo shutdown hook si mette in attesa su un oggetto, `shutdownSync`, per dare modo al thread main di chiudere il selector e terminare quindi l'esecuzione. Ho deciso di implementare questa routine di cleanup in modo da cercare, quando possibile, di garantire una chiusura "dolce" del programma chiudendo connessioni e thread e, nel caso in cui il flag `autoCommitDatabase` sia impostato su `false`, per tentare di eseguire un ultimo commit.

3 Client

Il client consiste in un'applicazione CLI mediante la quale interagire col server. Essendo Wordle un gioco, questa è stata sviluppata per essere il quanto più possibile immediata e coinvolgente per l'utente, pertanto è stato implementato l'uso di colori e di comandi molto semplici in modo che l'utente non debba ricordarsi i comandi testuali con i vari parametri. Una volta avviato il client, questo legge la configurazione, crea un oggetto `Client` e ne invoca il metodo `start()`: da questo punto in poi si entra in un ciclo infinito in cui viene letto l'input da tastiera e vengono mostrati menu e messaggi contestuali alle azioni dell'utente. La connessione del client al server avviene contestualmente alle operazioni effettuate: la connessione al servizio di registrazione mediante RMI viene effettuata non appena l'utente entra nel menu per la registrazione, mentre la connessione TCP viene instaurata non appena viene effettuato il login. Oltre alla connessione TCP, il client registra anche il servizio di callback per la notifica degli aggiornamenti in classifica e avvia un thread preposto alla ricezione di messaggi in arrivo sul gruppo multicast, i cui parametri (ip e porta) vengono recuperati dalla risposta del server alla richiesta di login. La classe `Requests` contiene le implementazioni dei metodi per effettuare richieste HTTP attraverso il `SocketChannel` tramite i metodi `get` e `post`. Per quanto riguarda la stampa dei messaggi e dell'interfaccia testuale, si è scelto un approccio più "modulare", rispetto alla semplice stampa di stringhe "hardcoded": nella classe `UserMessages` è contenuta una `HashMap` avente come chiave una stringa identificativa della lingua (sono state implementate l'italiano e l'inglese, rispettivamente `it-it` e `en-en`) e come valore un'altra mappa, che a sua volta ha come chiave una stringa mnemonica, comune a tutte le lingue, e come valore una stringa con il messaggio, già predisposto per la stampa "a colori" tramite la classe `CommonUtils.PrettyPrinter`. In questo modo è possibile andare a mantenere ed espandere l'interfaccia testuale col minimo sforzo.

3.1 La classe `Session`

Classe fondamentale dell'applicativo client, `Session` si occupa del salvataggio di tutte le informazioni della sessione di un utente: qui sono contenuti lo username dell'utente connesso, il token della sessione, se l'utente ha avviato una partita, il timestamp dell'ora a cui cambierà SW, i parametri di multicast e la lista delle notifiche ricevute. Ogni volta che il client si disconnette o l'utente effettua il logout, viene invocato il metodo `resetSession()` per azzerare i campi della sessione.

3.2 Il `multicastListener`

Il client, a differenza del server, attiva solo 2 thread: il thread main e il thread che si occupa della ricezione dei messaggi in arrivo sul gruppo multicast. Questo fa sì che l'unica struttura dati sulla quale viene eseguito l'accesso concorrente è una lista all'interno dell'istanza di `Session`, ovvero `shares`. I metodi che accedono a questa lista all'interno di `Session` sono:

- `addShare(String share)`: invocato dal thread `multicastListener` per aggiungere una nuova partita condivisa.
- `getNotificationNumber()`: durante la stampa del menu di gioco per mostrare quanti messaggi sono arrivati.
- `readNotifications()`: stampa a schermo le partite condivise, per poi svuotare la lista `shares`.

In tutti e tre i casi, l'accesso concorrente è garantito dall'uso di un monitor sulla lista `shares` stessa.

3.3 Uso

Una volta avviato il programma, si interagisce con i menu inserendo il numero dell'azione che si vuole eseguire e premendo "Enter". In particolare, prima di poter giocare è necessario eseguire il login, dopodiché si deve avviare una partita per poter iniziare a inviare delle GW. È possibile abilitare/disabilitare la stampa delle statistiche di gioco a ogni refresh[4] dell'interfaccia così da averle sempre sotto controllo. Per testare la vittoria, come visto nella sezione 2.1, è stato aggiunto al server un metodo per reperire la SW, così che sia sufficiente aprire sul browser l'url http://server-ip:server-port/getCurrentWord per poter avere la soluzione. L'utente di default creato durante la creazione di un nuovo database è `admin` con password `changeme`.

N.B.: Se si esegue il client su un terminale (non tramite un IDE), la password non sarà visibile durante la digitazione.

3.4 Gestione degli errori e shutdown hook

Le eccezioni sollevate all'interno del processo client sono, per lo più, legate a errori durante la comunicazione col server (`IOException` e `RemoteException` sono le più comuni), pertanto ho deciso di implementare un metodo, `handleConnectionException`, che ogniqualvolta una di queste eccezioni si verifica tenta la riconnessione col server per un numero di volte definito dall'utente nel file di configurazione. Nel caso la riconnessione non andasse a buon fine, viene invocato il metodo `System.exit()`. È stato inoltre registrato uno shutdown hook all'inizio dell'esecuzione del processo, così che quando l'utente esce dal programma, questo chiuda tutti i socket, interrompa il thread `multicastListener` ed esegua quello che viene definito un *graceful shutdown*, come nel caso del server.

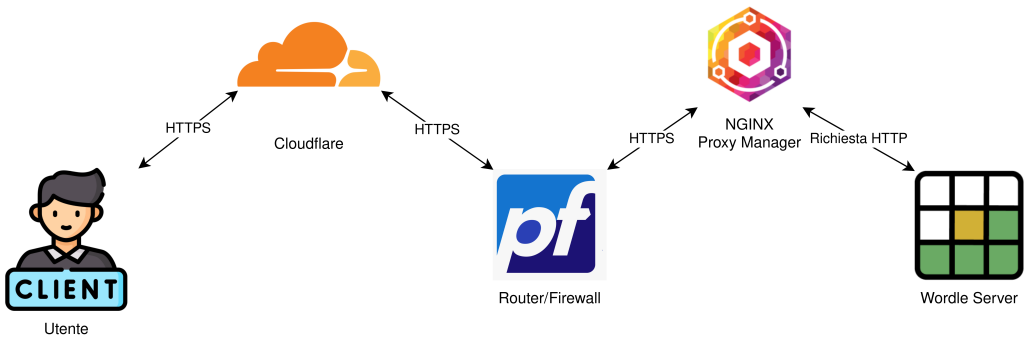
3.5 Dizionari per la traduzione della cli

Per poter modificare più facilmente l'applicazione client e poter facilmente implementare la traduzione dello stesso in più lingue, i messaggi che vengono stampati a schermo fanno parte di una hashmap che ha come chiave la lingua e come valore un'altra hashmap che a sua volta ha come chiave una stringa mnemonica e come valore i vari messaggi da stampare a schermo. In questo modo il client non conterrà delle stringe *hardcoded* e sarà pertanto più semplice andare a implementare nuove traduzioni o modificare i singoli messaggi.

4 L'interfaccia grafica

L'interfaccia grafica è stata realizzata in javascript come webapp utilizzando il popolarissimo framework React ed è hostata su un server privato. I certificati sono rilasciati da Let's Encrypt, mentre la comunicazione HTTPS viene instaurata fra il client e il reverse proxy, mentre fra il reverse proxy e il server viene usato il protocollo HTTP.

Di seguito uno schema del setup:

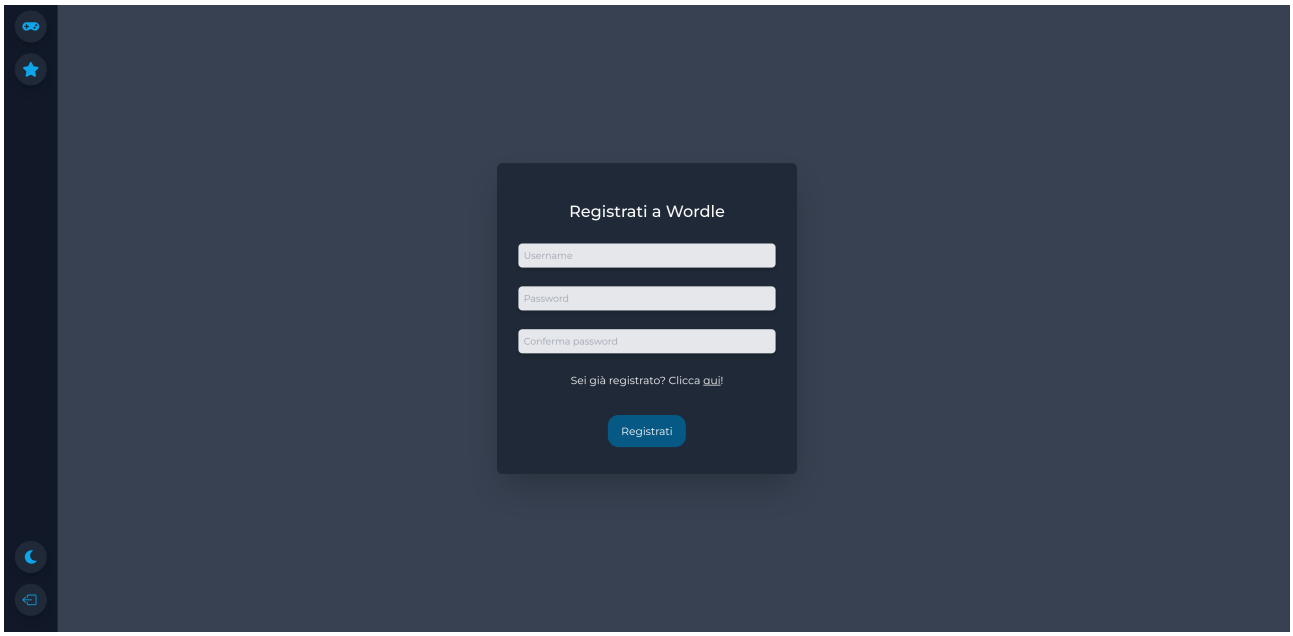


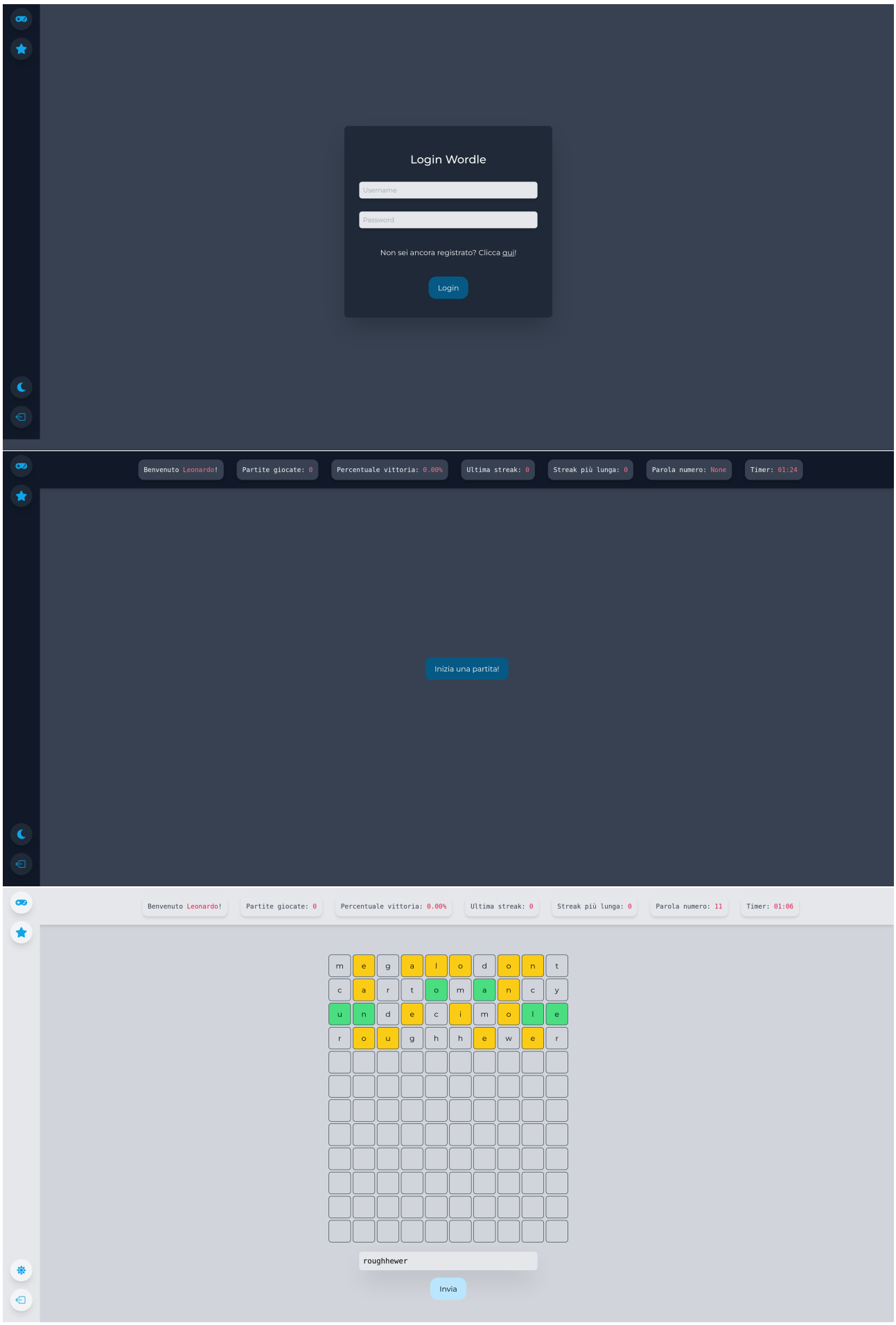
[4] è possibile eseguire un refresh della CLI premendo "Enter" senza digitare alcun numero

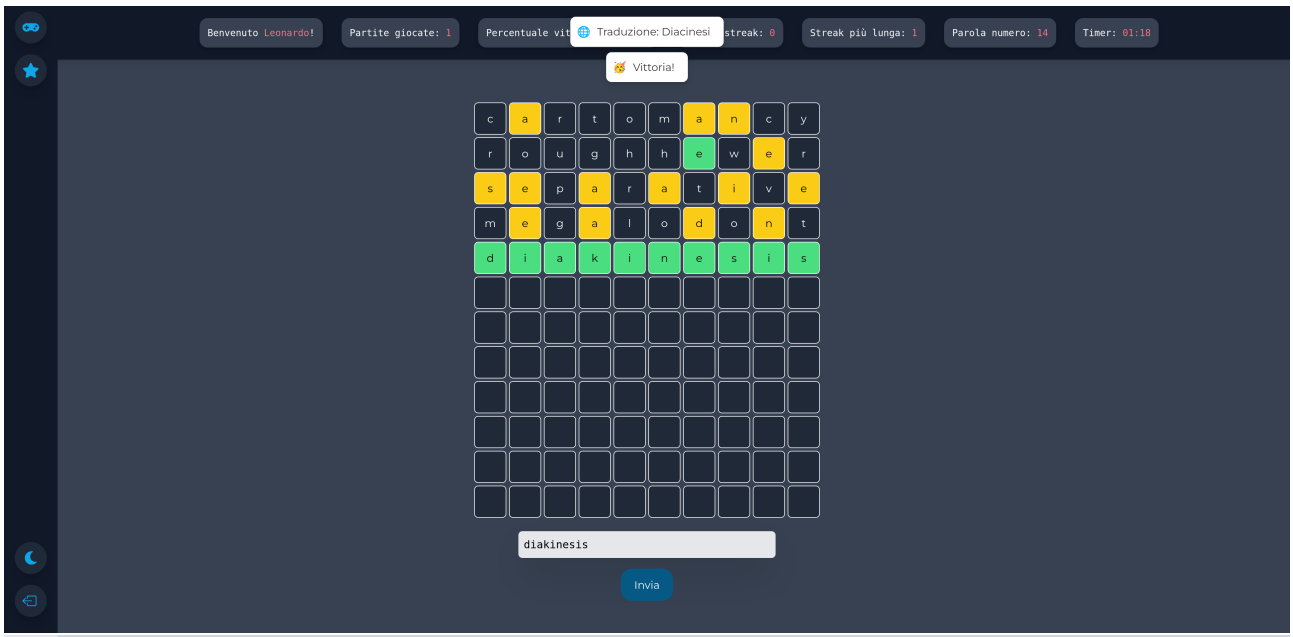
Allegato al progetto vi è solo il codice sorgente, mentre per testarla è possibile collegarsi al link `https://wordle.alteregofiere.com`, così che non siano necessari ulteriori step di configurazione. La configurazione è la seguente:

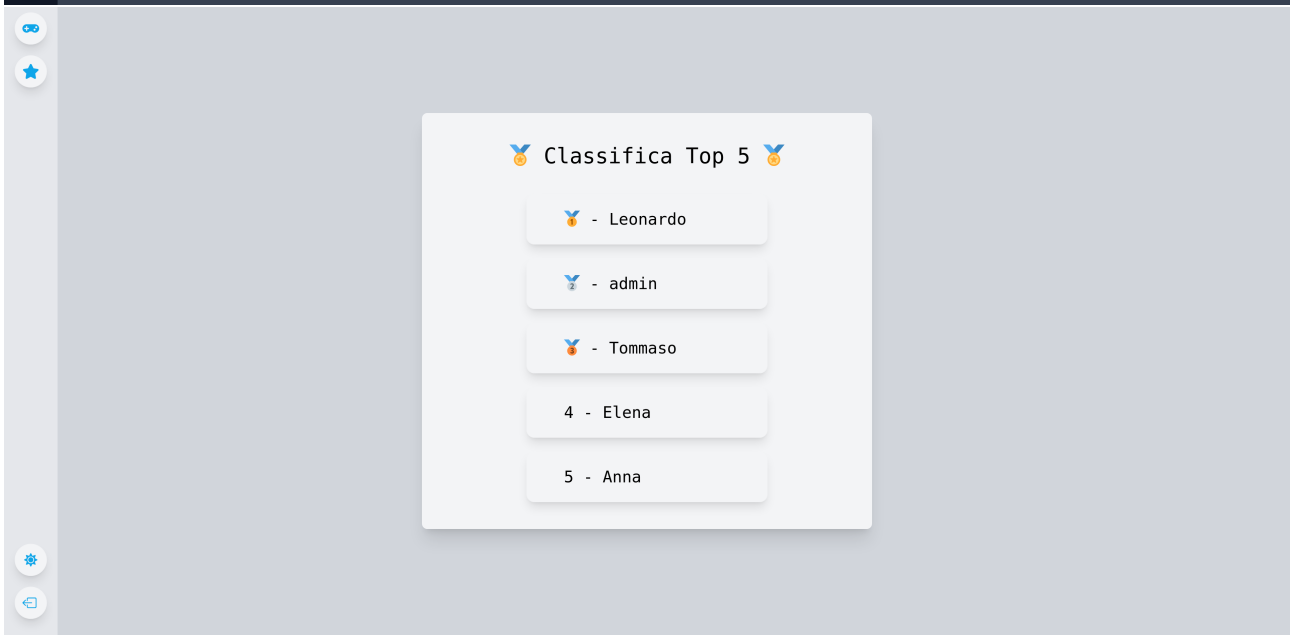
```
{
  "tcpPort" : 8801,
  "rmiPort" : 3800,
  "multicastPort" : 7800,
  "multicastAddress" : "230.0.0.1",
  "corePoolSize" : 4,
  "secretWordTimeout" : 120000,
  "verbose" : true,
  "sqliteDatabasePath" : "~/Wordle/Database/sqlite.db",
  "autoCreateDatabase" : true,
  "autoCommitDatabase" : false,
  "wordsFilePath" : "~/Wordle/Wordle_Project/words.txt",
  "useJsonDatabase" : true,
  "jsonDatabasePath" : "~/Wordle/Database/Json",
  "debug" : true
}
```

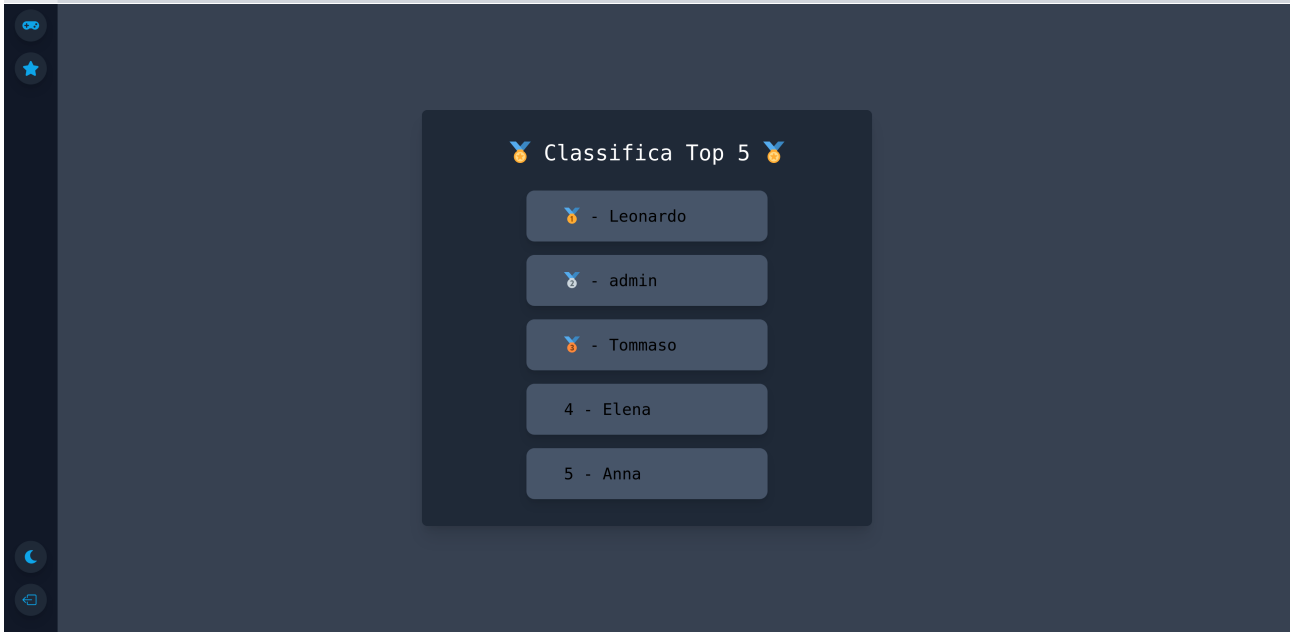
È possibile anche in questo caso usare l'utente di default `<admin, changeme>`. Essendo il flag `debug` settato su `true`, è possibile reperire la SW al link `https://wordle.alteregofiere.com/getCurrentWord`. Di seguito alcuni screen:




Login Wordle
Username
Password
Non sei ancora registrato? Clicca qui!
Login
Benvenuto Leonardo!
Partite giocate: 0
Percentuale vittoria: 0.00%
Ultima streak: 0
Streak più lunga: 0
Parola numero: None
Timer: 01:24
Inizia una partita!
Benvenuto Leonardo!
Partite giocate: 0
Percentuale vittoria: 0.00%
Ultima streak: 0
Streak più lunga: 0
Parola numero: 11
Timer: 01:06
m e g a l o d o n t
c a r t o m a n c y
u n d e c i m o l e
r o u g h h e w e r
roughhewer
Invia


Benvenuto Leonardo!
Partite giocate: 1
Percentuale vit
Traduzione: Diacinesi
streak: 0
Streak più lunga: 1
Parola numero: 14
Timer: 01:18
Vittoria!
diakinesis
Invia


Classifica Top 5
- Leonardo
- admin
- Tommaso
4 - Elena
5 - Anna


Classifica Top 5
- Leonardo
- admin
- Tommaso
4 - Elena
5 - Anna