Laboratorio di Reti, Corso A

Word Quizzle (WQ)

Progetto di Fine Corso

A.A. 2019/20

Gazzarrini Marina

531162

14 Febbraio 2020

Indice

Introduzione

Il progetto consiste nell' implementazione di un gioco il cui obbiettivo è accumulare punti attraverso delle sfide di traduzione italiano-inglese.
Gli utenti possono registrarsi e, dopo aver effettuato il login, usufruire del servizio. Possono anche aggiungere amici, visionare la classifica e il proprio punteggio. Per utilizzare tale servizio sono messi a disposizione vari comandi con cui poter effettuare le richieste:

- ***registra_utente*** *nome password*: per potersi registrare al servizio. È necessario specificare nome utente e password, le stesse che in seguito potranno essere usate per effettuare il login.

- ***login*** *nome password*: specificando nome utente e password dà la possibilità di effettuare il login ed accedere al proprio profilo.

- ***logout***: effettua il logout dell'utente.

- ***aggiungi_amico*** *nomeamico*: per poter aggiungere un amico alla propria lista è sufficiente specificare il nome dell'amico.

- ***lista_amici***: consente di vedere la lista dei propri amici.

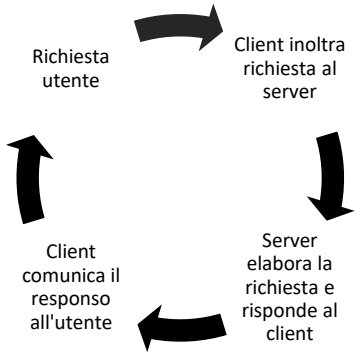
- ***mostra_punteggio***: per prendere visione del punteggio complessivo ottenuto da tutte le sfide effettuate.

- ***mostra_classifica***: consente di vedere la classifica (nome e punteggio) dei propri amici.

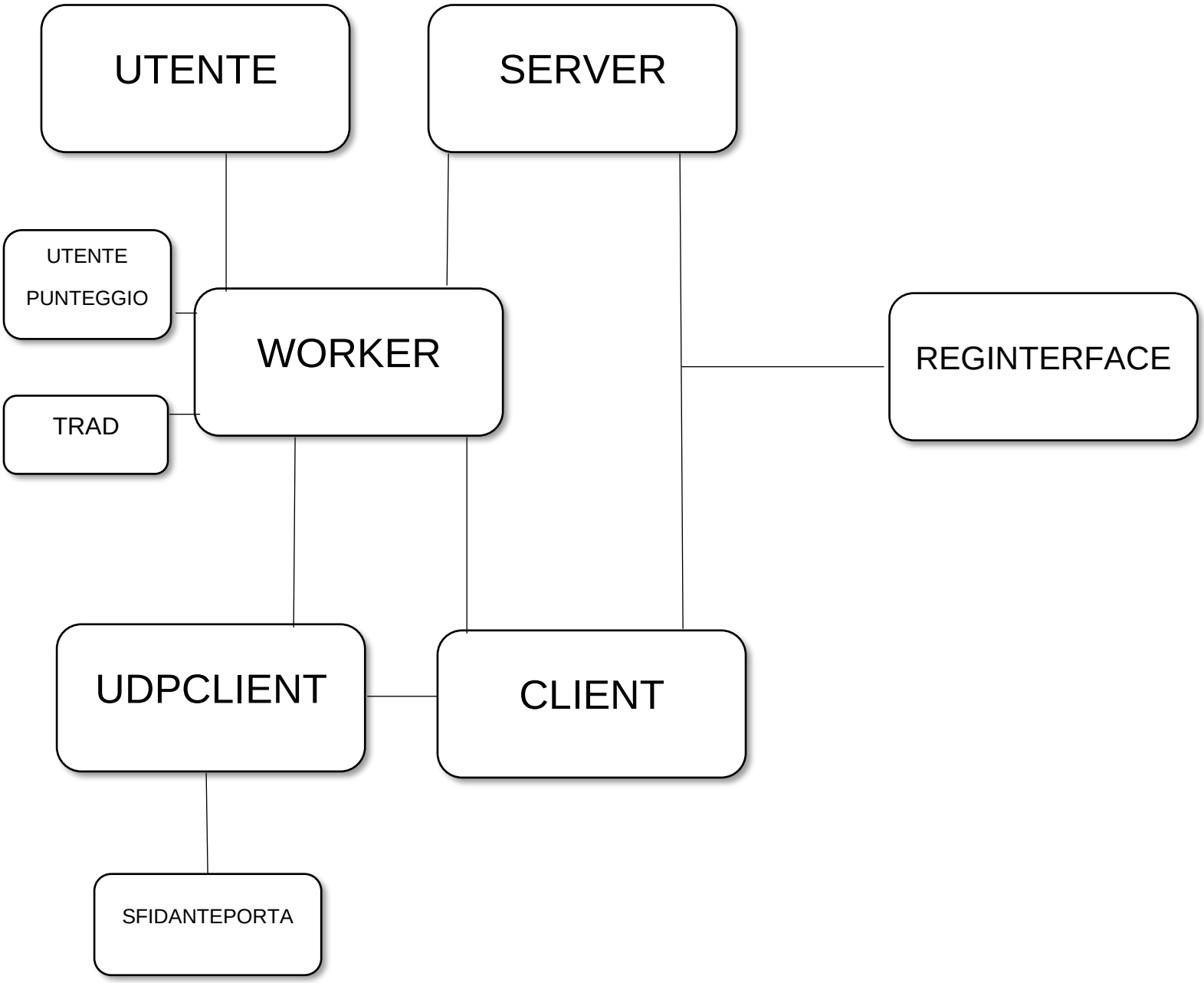
- ***Sfida*** *nomeamico*: fa partire una richiesta di sfida nei confronti dell'amico specificato.

- ***mostra_richiesta_sfida***: per poter vedere la richiesta di sfida meno recente ed accettarla (o eventualmente rifiutarla).

- ***wq --help:*** mostra la lista dei comandi possibili.



Schema generale classi



Il server gestisce la comunicazione TCP e UDP con i client attraverso dei *thread*: un thread per ogni client. Utentepunteggio, trad e sfidanteporta sono delle classi di appoggio il cui funzionamento verrà spiegato in seguito.

Classi

1. Server

Il server è stato realizzato mediante *multithreaded*. Durante la fase di inizializzazione legge se presente il file "Backup", salvato nella directory di lavoro e contenente tutte le informazioni degli utenti da persistere, e ne salva il contenuto in una ConcurrentHashMap così che sia più veloce reperire in seguito tali informazioni. Successivamente si occupa di inserire le parole presenti nel file "Dizionario" (anche esso presente nella directory di lavoro) in un'altra ConcurrentHashMap che verrà utilizzata durante le sfide. Si occupa anche della registrazione che è stata implementata mediante RMI.
Per gestire le interazioni con il client utilizza dei thread, più precisamente una *CachedThreadPool* che permette la creazione di un pool con un comportamento predefinito: se tutti i thread sono occupati ne viene creato uno nuovo, altrimenti viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente. Questo tipo di threadpool è stata scelta principalmente perché fornisce un buon livello di elasticità. Viene attivato un thread per ogni client e tale thread si occuperà di gestire l'intera comunicazione con quel client.
In alternativa a questa implementazione, si poteva scegliere di usare un *selector* così che fosse possibile la gestione con lo stesso thread di più eventi che avvengono simultaneamente; è stato scelto di non farlo per una maggiore *"responsiveness".*

2. regInterface

Questa classe implementa l'interfaccia per la ricezione delle richieste di registrazione da parte dei client, utilizzando il *servizio RMI*. Al momento della registrazione si verifica che il nome fornito dall'utente non sia già presente e viene notificato se la registrazione è andata a buon fine o meno. In caso di successo il nome e la password vengono salvati sia in una ConcurrentHashMap, che all'interno di un file Backup poiché sono informazioni da persistere.

3. Utente

Questa classe implementa *Serializable* e contiene tutte le informazioni relative ad un utente, sia quelle da persistere (e che verranno quindi anche salvate in un apposito file json) che quelle necessarie durante l'esecuzione del servizio. Contiene il nome, la password, il punteggio totale, la lista degli amici, se l'utente è online o meno, se è in sfida e altre informazioni temporanee necessarie durante il corso

della sfida. Contiene anche tutti i metodi necessari per reperire e modificare tali informazioni. Per ogni utente, al momento della registrazione viene creata un'istanza della classe Utente e viene inserita come valore all'interno di una ConcurrentHashMap, che come chiave avrà invece il nome dell'utente stesso.

4. utentePunteggio

Si tratta semplicemente di una classe di appoggio creata per *facilitare l'ordinamento* degli amici all'interno della classifica. Implementa *Comparable* e tiene conto del nome di un utente e del proprio punteggio.

5. trad

Anche questa è una classe di appoggio ed è utilizzata per facilitare il recupero delle traduzioni per la sfida. Ogni istanza contiene un nome e la rispettiva traduzione (all'inizio quest'ultima non sarà presente, verrà aggiunta in seguito dopo essere stata recuperata attraverso una chiamata HTTP GET). Sarà creata un'istanza di trad per ogni parola presente nel file Dizionario e diventerà il valore del campo di una ConcurrentHashMap che ha la funzione di "cache" per quanto riguarda le parole già tradotte.

6. Worker

Classe che implementa *Runnable* e si occupa della comunicazione con i client, sia attraverso tcp che udp. Implementa e gestisce tutte le varie operazioni che possono essere richieste dall'utente: legge le richieste, le elabora e restituisce una risposta al client. Tutto ciò avviene facendo uso di una *connessione TCP*. Quando arriva una richiesta di login controlla che password e nome utente siano corretti e comunica il responso all'utente, così come nel caso di logout. Se si richiede di aggiungere un nuovo amico, viene modificata la lista amici sia dell'utente che effettua la richiesta, sia dell'amico (sia nella ConcurrentHashMap che nel file json). Restituisce la lista degli amici (è stata utilizzata una lista così come da richiesta, ma personalmente avrei ritenuto più efficiente un'implementazione con una HashMap a causa delle frequenti ricerche) e il punteggio corrente andando a recuperare le informazioni direttamente dalla ConcurrentHashMap e si occupa anche della creazione e della restituzione della classifica di ogni utente.
Infine si occupa dell'inoltro della richiesta di sfida utilizzando udp e anche dell'implementazione della sfida stessa, i cui dettagli verranno discussi in seguito.

7. Client

Ogni client legge da console le richieste dell'utente in modo interattivo e le trasferisce al server utilizzando una connessione TCP. Dopo di che stampa a schermo le risposte del server, il tutto all'interno di un ciclo. Lancia anche un thread che si occupa della gestione della ricezione di messaggi UDP provenienti dal server.
Contiene una sezione dedicata alla sfida a cui si accede dopo aver accettato una richiesta di sfida attraverso UDP.

8. udpClient

Implementa Runnable e si occupa della *ricezione UDP* di messaggi da parte del server. In un ciclo attende messaggi (ovvero nuove richieste di sfida) e li inserisce all'interno di una lista thread safe a cui ha accesso condiviso con il client. Ogni elemento della lista è un'istanza della classe sfidantePorta e contiene il nome dello sfidante, la porta da cui è arrivato il messaggio e il tempo di arrivo (necessario poiché c'è un limite di tempo alla validità delle richieste di sfida).

9. sfidantePorta

Si tratta di una classe di appoggio contenente informazioni relative alle richieste di sfida; ovvero il nome dello sfidante, la porta da cui proviene il messaggio (necessaria poi per rispondere alla richiesta) e il tempo di arrivo del messaggio. Queste informazioni verranno conservate in una lista *thread safe* e recuperate dal client così da poter accettare o rifiutare le varie proposte di sfida.

Strutture

Le strutture maggiormente rilevanti all'interno del progetto sono tre: due ConcurrentHashMap e una LinkedBlockingQueue.

- *ConcurrentHashMap dati*: lato server viene utilizzata per salvare tutte le informazioni rilevanti a proposito degli utenti. La chiave è una stringa che corrisponde al nome di un utente, il valore è un'istanza della classe Utente contenente tutti i dettagli corrispondenti. Al momento dell'avvio del server vengono copiati all'interno di "dati" tutte le informazioni presenti nel file di backup e poi viene costantemente aggiornata. Viene utilizzata poiché è più veloce e semplice reperire informazione da qui piuttosto che dal file json.

- *ConcurrentHashMap cache*: lato server viene utilizzata per salvare le parole presenti nel file Dizionario ed eventualmente le rispettive traduzioni. Contiene come chiavi degli interi (da 1 a 446, ovvero il numero totale delle parole presenti nel file) e come valore un'istanza della classe trad (quindi una coppia di stringhe <termine italiano, traduzione inglese>). Inizialmente conterrà solo i termini in italiano, le traduzioni saranno aggiunte man mano che verranno recuperate nel corso delle varie sfide. È stato scelto di inserire le parole all'interno di questa struttura e non utilizzare direttamente il file perché sarebbe stato più scomodo e costoso.

- *LinkedBlockingQueue messaggi:* lato client viene utilizzata per salvare informazioni a proposito dei messaggi UDP ricevuti dal server. Contiene istanze della classe sfidantePorta, una per ogni messaggio.

Tutte e tre le strutture sono thread safe poiché c'è la possibilità che più thread operino sulla stessa struttura in contemporanea generando degli stati inconsistenti.

Gestione della concorrenza

Lato client, oltre al thread che gestisce il client, viene attivato un thread per gestire la ricezione di messaggi su UDP.
Lato server, oltre al thread che gestisce il server, si usa una CachedThreadPool.

Come specificato anche precedentemente le strutture dati utilizzate sono delle *Concurrent Collections* che implementano meccanismi di lock implicite. Inoltre i metodi che operano su oggetti condivisi da più thread sono stati dichiarati *synchronized*.

Informazioni da persistere e file json

Le informazioni da persistere su un file json sono le informazioni di registrazione (nome e password), le relazioni di amicizia (la lista degli amici) e il punteggio. Per fare ciò viene creato, se non è presente (se lo è viene semplicemente letto), un file json di backup al momento di avvio del server. Questo file viene aggiornato ogni volta che viene apportata una modifica ad uno dei campi sopra citati così da garantire che le informazioni non vadano mai perse. Per l'aggiornamento del file viene utilizzato un opportuno metodo dichiarato synchronized.
Per ogni utente, viene salvato all'interno del file un JSON object contenente un array (con all' interno gli amici dell'utente) e 3 coppie chiave-valore per salvare il nome dell'utente, il punteggio e la password.

Librerie esterne

Come unica libreria esterna è stata usata *JSON-simple* per la gestione di oggetti json. È stata preferita rispetto ad altre librerie poiché è semplice e leggera.

La sfida

Ogni sfida può avvenire solo fra due utenti amici ed è implementata usando TCP. Tramite il comando “sfida” è possibile inoltrare la richiesta di sfida ad un proprio amico che, a sua volta, avrà la possibilità di accettarla o meno. L’inoltro della richiesta di sfida dal server all’amico sfidato avverrà attraverso UDP, come anche la risposta di quest’ultimo. Per capire meglio l’implementazione della sfida verrà mostrato un esempio, esplicitando ogni passaggio.

Utente A vuole sfidare un proprio amico utente B; per fare ciò invia sulla connessione TCP una richiesta “sfida B”. A quel punto il thread che si occupa del client A (che chiameremo ThreadA) controlla che B faccia parte della lista amici, che non sia in sfida e che sia online; se anche solo una di queste condizioni non è verificata manda un messaggio di errore ad A. Se tutto va bene invece il ThreadA procede con l’inoltro della richiesta di sfida a B: attraverso UDP, usando la stessa porta che B usa per la connessione TCP (questa informazione viene salvata per ogni utente), invia il messaggio e setta un timer. Se il timer scade oppure B risponde negativamente, viene inviato un messaggio ad A, se la risposta è positiva (utente deve digitare “si”) inizia la preparazione della sfida. ThreadA sceglie casualmente (utilizza gli interi usati come chiave nella ConcurrentHashMap) le parole per la sfida e cerca le traduzioni (prima nella ConcurrentHashMap “cache”, se non presenti chiede tramite una chiamata HTTP GET la traduzione al servizio esterno accessibile mediante la URL https://mymemory.translated.net/doc/spec.php e salva i termini tradotti anche in “cache”), inserisce tutto in un array che verrà condiviso con ThreadB che si occuperà della sfida interagendo con B. Durante la sfida controlla che il tempo limite per le traduzioni non venga superato e assegna punti in base alle traduzioni corrette.

Passiamo ora ad analizzare il comportamento di B. Le richieste di sfida verso il client B sono contenute all’interno di una LinkedBlockingQueue e possono essere recuperate una alla volta in due modi: utilizzando il comando “mostra_richiesta_sfida” che mostra la richiesta di sfida meno recente (e non scaduta) oppure, alla fine di ogni risposta del server (a qualsiasi richiesta, che sia di login o di mostrare il punteggio ecc…), compare la richiesta di sfida più vecchia. A questo punto B può accettare o meno la sfida, in ogni caso manda un messaggio di risposta usando UDP al ThreadA. Nel caso accetti, passa alla sezione “in sfida”: invia in automatico una stringa al thread con cui comunica usando TCP (ThreadB) per informarlo dell’inizio della sfida e poi procede tutto regolarmente come scritto sopra.

Le perdite per quanto riguarda UDP vengono gestite usando dei timer.

Esecuzione

Provato sul sistema operativo Windows 10 Versione 1903 (build SO 18362,592) usando Eclipse Photon June 2018.

server: è la classe da mettere in esecuzione per attivare il server.

client: è la classe da mettere in esecuzione per attivare il client.

Il sistema è stato progettato per funzionare sull'interfaccia *localhost*, ma per quanto riguarda le traduzioni dall'italiano all'inglese si serve di un servizio esterno ed è quindi dipendente dalla connessione di rete.

Lista richieste e testing

registra_utente:

Fase di registrazione implementata mediante RMI. Per prima cosa il client controlla se l'utente che chiede di effettuare la registrazione è online o meno, se lo è stampa un messaggio di errore (non si può registrare nuovi utenti se abbiamo effettuato il login). Successivamente controlla che il formato della richiesta sia corretto, ovvero che siano stati inseriti altri due termini (che corrisponderanno a nome utente e password); se tutto è ok invia un messaggio di richiesta registrazione al server mediante connessione TCP.
Il server appena ricevuta la richiesta controlla se il nome fornito è già in uso, in quel caso restituisce un codice di errore (2), altrimenti inserisce il nuovo utente e la rispettiva password sia nella *ConcurrentHashMap* dati che nel *file Backup*.

Testing:

- Registrazione utente avvenuta con successo
- Tentativo di registrazione con nome utente già in uso
- Tentativo di registrazione con un formato errato
- Tentativo registrazione dopo aver effettuato il login

```
registra_utente luigi sissi
registrazione avvenuta con successo
registra_utente marina sissi
username già in uso, registrazione fallita
registra_utente luca
Impossibile effettuare la registrazione poichè il formato della richiesta non è corretto
login luigi sissi
Login effettuato con successo
registra_utente luca sissi
Impossibile effettuare la registrazione poichè si è già loggati
```

login:

Per richiedere un'operazione di login è necessario specificare nome utente e password. Lato client si controlla che l'utente non sia già loggato e che il formato della richiesta sia corretto, poi si invia una richiesta al server mediante connessione TCP.
Lato server si controlla che l'utente sia registrato (se non lo è si manda come codice di errore 3), che la password inserita sia corretta (se non lo è il codice restituito è 2) e che non sia già online su un altro dispositivo (codice 4). Per effettuare questi controlli si utilizzano le informazioni salvate nella *ConcurrentHashMap*. Nel caso tutto proceda correttamente si restituisce il valore 1.

Testing:

- Tentativo login utente non registrato
- Tentativo login con password errata
- Tentativo login formato errato
- Tentativo login utente già online su un altro dispositivo
- Login effettuato con successo
- Tentativo di login quando abbiamo già effettuato precedentemente il login

```
login luca sissi
L'utente non è registrato
login luigi si
La password inserita non è corretta
login luigi
Impossibile effettuare il login poichè il formato della richiesta non è corretto
login marina luna
L'utente è già online su un altro dispositivo
login luigi sissi
Login effettuato con successo
login luigi sissi
Impossibile effettuare il login poichè già online
```

aggiungi_amico:

È necessario specificare il nome dell'amico che si intende aggiungere alla propria lista. L'aggiunta è *"bidirezionale"*, automaticamente anche l'utente che fa la richiesta viene aggiunto alla lista degli amici dell'amico.
Lato client si controlla che l'utente sia loggato e che il formato della richiesta sia corretto e si comunica al server.
Lato server si controlla, utilizzando la *ConcurrentHashMap*, che l'amico sia registrato (codice errore 3) e che non sia già presente nella lista degli amici (codice errore 2). Se tutto procede correttamente si aggiorna la lista degli amici sia sul file JSON, poiché è un'informazione da persistere, sia nella ConcurrentHashMap (sia la lista dell'utente che fa la richiesta, sia la lista dell'amico).

Testing:

- Richiesta aggiunta amico non registrato
- Formato richiesta errato
- Richiesta corretta
- Richiesta aggiunta amico già presente nella lista amici

```
aggiungi_amico luca
Impossibile aggiungere luca agli amici perchè non è registrato
aggiungi_amico
Richiesta non riconosciuta
aggiungi_amico erika
erika aggiunto agli amici
aggiungi_amico erika
erika era già presente tra gli amici
```

lista_amici:

Lato client si controlla che l'utente che sta effettuando la richiesta abbia precedentemente effettuato il login, poi si inoltra la richiesta al server. Il server recupera la lista degli amici, usando la ConcurrentHashMap, e poi ottiene un oggetto json.

Testing:

- Richiesta lista amici senza aver effettuato il login
- Lista amici con più elementi
- Lista amici vuota

```
lista_amici
Prima è necessario effettuare il login
login luigi sissi
Login effettuato con successo
lista_amici
Lista amici: erika
```

```
login leonardo stella
Login effettuato con successo
lista_amici
Nessun amico
```

mostra_punteggio:

Il client controlla che l'utente sia online al momento della richiesta e comunica con il server mediante TCP. Il server reperisce il punteggio totale delle sfide andando a controllare nella ConcurrentHashMap e lo restituisce al client.

Testing:

- Richiesta senza aver effettuato il login
- Richiesta corretta

```
mostra_punteggio
Prima è necessario effettuare il login
login marina luna
Login effettuato con successo
mostra_punteggio
Il punteggio: 25
```

mostra_classifica:

Come di consueto il client controlla che prima di fare la richiesta l'utente abbia effettuato il login, poi inoltra la richiesta al server. Il server recupera la lista degli amici e i rispettivi punteggi (utente compreso) usando la *ConcurrentHashMap*, dopo di che procede a stilare una classifica facendo uso di una classe utentePunteggio che implementa *Comparable*. Così facendo ordina i componenti in base al punteggio (ordine decrescente), incapsula il tutto in un oggetto json e lo invia al client.

Testing:

- Richiesta senza aver effettuato il login
- Richiesta corretta

```
mostra_classifica
Prima è necessario effettuare il login
login luigi sissi
Login effettuato con successo
mostra_classifica
Classifica: marina:25, erika:18, luigi:0,
```

mostra_richiesta_sfida:

Consente di prendere visione della richiesta di sfida meno recente (ancora valida) e di accettarla o rifiutarla. La richiesta può essere effettuata direttamente dall'utente da console, ma viene anche eseguita automaticamente al termine di ogni altra richiesta (login, mostra_punteggio, aggiungi_amico, mostra_classifica, lista_amici) da parte dell'utente. È stato scelto di poter prendere visione di una sola richiesta alla volta a partire dalla meno recente per dare priorità all'ordine con cui le richieste di sfida sono state inviate. Questo metodo è interamente gestito dal client: per prima cosa controlla se la richiesta proviene direttamente dall'utente o è stata formulata in automatico. Nel secondo caso stampa a schermo solo se è presente una richiesta di sfida (così che l'utente non percepisca ogni volta questo controllo non richiesto da lui). Nel primo caso invece stampa a schermo anche la mancata presenza di richieste di sfida. Tutte le richieste pendenti sono contenute in una LinkedBlockingQueue: si scorre la lista all'interno di un ciclo, si eliminano le richieste di sfida che non sono più valide (poiché è trascorso troppo tempo) e si visualizza la richiesta meno recente in modo che l'utente possa accettarla o meno. Il client invia il responso al server tramite UDP.

Testing:

- Richiesta senza aver effettuato il login
- Richiesta esplicita quando non sono presenti richieste pendenti
- Richiesta esplicita quando è presente almeno una richiesta pendente
- Richiesta implicita quando è presente almeno una richiesta pendente

```
mostra_richiesta_sfida
prima effettuare il login
login luigi sissi
Login effettuato con successo
mostra_richiesta_sfida
Nessuna richiesta di sfida
mostra_richiesta_sfida
Hai una nuova richiesta di sfida da parte di erika
Accetti?
no
Hai rifiutato la sfida
mostra_punteggio
Il punteggio: 0
Hai una nuova richiesta di sfida da parte di marina
Accetti?
no
Hai rifiutato la sfida
```

sfida:

È necessario specificare l'amico che si vuole sfidare. Il client controlla che l'utente che richiede la sfida abbia effettuato il login e inoltra la richiesta al server. Il server controlla che l'amico che si vuole sfidare online, presente nella lista degli amici e non attualmente in sfida. In seguito inoltra la richiesta di sfida e procede come spiegato in precedenza.

Testing:

- Richiesta sfida senza aver effettuato il login
- Richiesta sfida formato errato
- Richiesta sfida amico non online, già in sfida o non presente nella lista
- Richiesta sfida corretta e rifiutata
- Richiesta sfida corretta e accettata

```
sfida marina
Prima è necessario effettuare il login
login luigi sissi
Login effettuato con successo
sfida
Richiesta non riconosciuta
sfida carlo
Impossibile effettuare la sfida: amico non presente nella lista o non online
sfida marina
Sfida non accettata
sfida marina
Sfida accettata
La sfida ha inizio!
```

logout:

In questo caso l'unico controllo effettuato lato client è che l'utente sia loggato per poter effettuare il logout. Poi si comunica la richiesta al server.

Testing:

- Tentativo logout senza aver effettuato prima il login
- Logout corretto

```
logout
Non si può effettuare il logout prima di aver effettuato il login
login luigi sissi
Login effettuato con successo
logout
Logout effettuato con successo
```

Esempio esecuzione e sfida

Per mostrare un esempio di sfida si fa vedere direttamente una schermata del computer.

Informazioni iniziali file json:

File Modifica Formato Visualizza ?

```
{"Amici":[{"Amico":"veronica"}],"Punteggio":25,"Nome":"marina","Password":"luna"}
{"Amici":[],"Punteggio":0,"Nome":"luigi","Password":"sissi"}
{"Amici":[],"Punteggio":0,"Nome":"erika","Password":"argo"}
{"Amici":[{"Amico":"marina"}],"Punteggio":18,"Nome":"veronica","Password":"macchia"}
```

Esempio di sfida e di alcune richieste:

client 1:

```
start client

LISTA COMANDI:
registra_utente nome password ->per effettuare la registrazione
login nome password ->per effettuare il login
logout ->per effettuare il logout
aggiungi_amico nomeAmico ->creare una relazione di amicizia con nomeAmico
lista_amici ->mostra la lista dei propri amici
mostra_punteggio ->restituisce il punteggio totale
mostra_classifica ->mostra la classifica degli amici
sfida nomeAmico->mandare una richiesta di sfida
mostra_richiesta_sfida-> mostra la richiesta di sfida meno recente

login marina luna
Login effettuato con successo
lista_amici
Lista amici: veronica
aggiungi_amico luigi
luigi aggiunto agli amici
lista_amici
Lista amici: veronica luigi
mostra_punteggio
Il punteggio: 25
mostra_classifica
Classifica: marina:25, veronica:18, luigi:0,
sfida luigi
Sfida accettata
La sfida ha inizio!
Parola 1/5
ospedale
hospital
Parola 2/5
dire
talk
Parola 3/5
lingua
tongue
```

```
Parola 4/5
bravo
good
Parola 5/5
successo
success
fine sfida attendi i risultati!
Sfida terminata! Punteggio ottenuto: -3 Punteggio avversario: -5 Totale parole corrette: 1
mostra_punteggio
Il punteggio: 22
mostra_classifica
Classifica: marina:22, veronica:18, luigi:-5,
```

client 2

```
start client

LISTA COMANDI:
registra_utente nome password ->per effettuare la registrazione
login nome password ->per effettuare il login
logout ->per effettuare il logout
aggiungi_amico nomeAmico ->creare una relazione di amicizia con nomeAmico
lista_amici ->mostra la lista dei propri amici
mostra_punteggio ->restituisce il punteggio totale
mostra_classifica ->mostra la classifica degli amici
sfida nomeAmico->mandare una richiesta di sfida
mostra_richiesta_sfida-> mostra la richiesta di sfida meno recente

login luigi sissi
Login effettuato con successo
lista_amici
Lista amici: marina
mostra_punteggio
Il punteggio: 0
Hai una nuova richiesta di sfida da parte di marina
Accetti?
si
La sfida ha inizio!
Parola 1/5
ospedale
hospital
Parola 2/5
dire
to talk
Parola 3/5
lingua
tongue
Parola 4/5
bravo
good
Parola 5/5
successo
succes
fine sfida attendi i risultati!
Sfida terminata! Punteggio ottenuto: -5 Punteggio avversario: -3 Totale parole corrette: 0
mostra_punteggio
Il punteggio: -5
mostra_classifica
Classifica: marina:22, luigi:-5,
```

Esempio di sfida timer scaduto:

```
mostra_classifica
Classifica: marina:22, veronica:18, luigi:-5,
mostra_richiesta_sfida
Hai una nuova richiesta di sfida da parte di veronica
Accetti?
si
La sfida ha inizio!
Parola 1/5
frase
phrase
Parola 2/5
bollente
hot
Parola 3/5
vecchio
old
Parola 4/5
Tempo scaduto!
Sfida terminata! Punteggio ottenuto: 1 Punteggio avversario: 1 Totale parole corrette: 2
mostra_punteggio
Il punteggio: 23
logout
Logout effettuato con successo
```

Note

La cartella compressa “531162_GAZZARRINI_MARINA” oltre ai sorgenti contiene anche due file: Backup e Dizionario.

È stato scelto di lasciare il file *Backup* con delle informazioni sugli utenti già registrati nel caso si voglia fare dei test senza dover perdere del tempo a registrare utenti e stabilire relazioni di amicizia. È però possibile eliminare il file e ripartire totalmente da zero.

Il file *Dizionario* contiene le parole da poter tradurre durante le sfide e non deve essere eliminato.