Relazione Word Quizzle (WQ)

Mariagiovanna Rotundo

560765

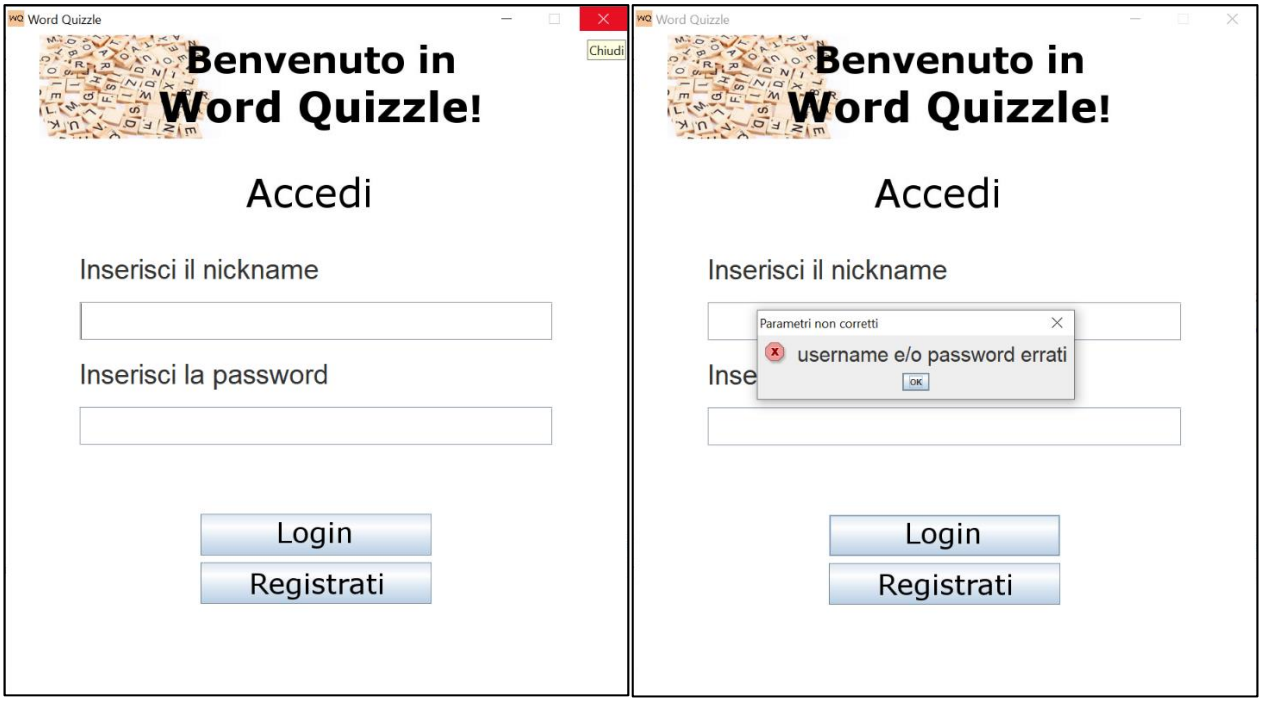
Indice

Descrizione generale dell'architettura

Interazione con l'utente: Interfaccia grafica e controlli

È stato scelto di far interagire l'utente con un'interfaccia grafica, piuttosto che un'interfaccia a linea di comando, che limita i possibili comandi mandati al server.

È stato deciso di fare alcuni controlli sui parametri inseriti dall'utente direttamente nel client così che, se i parametri non rispettano delle determinate caratteristiche, il server non elabora una richiesta che sicuramente fallirà in quanto invia parametri invalidi. Così si è cercato di alleggerire il carico di lavoro del server e di rendere l'applicazione un po' più scalabile. I controlli che necessitano accesso ai file che rappresentano i database e quindi mantengono anche informazioni di altri clienti vengono eseguiti lato server.



Quando l'utente avvia il programma gli viene mostrata la schermata nella figura in alto a sinistra. Qui sono permesse all'utente 4 operazioni: la chiusura dell'applicazione, la riduzione a icona della finestra, il login e la registrazione. La finestra ha dimensione fissa e non è permessa l'operazione di ridimensione.

La chiusura dell'applicazione, che può essere richiesta cliccando sulla croce in alto a destra nella schermata (rossa nell'immagine in alto a sinistra), non richiede controlli o comunicazione con il server in quanto l'utente non si è ancora connesso ad esso.

La riduzione a icona, che può essere richiesta cliccando su "-" in alto a destra nella schermata, non richiede mai comunicazione con il server in quanto non dipende in nessun modo da esso. Questa operazione è permessa in ogni schermata, anche successive a quella appena mostrata e non sarà ripetuta nelle successive schermate come possibile operazione.

Invece, quando l'utente richiede le operazioni di login o registrazione vengono fatti dei controlli prima di utilizzare l'oggetto remoto nel server, nel caso della registrazione, o di connettersi al server, nel caso di login.

Guardando anche esempi di applicazione, è stato scelto di limitare il numero di caratteri di nickname e password validi a 20. Questa non è vista come una limitazione visto l'elevato numero di combinazioni possibili. Gli spazi vuoti non vengono considerati e vengono rimossi in maniera trasparente all'utente. Nickname e password sono case sensitive.

Al momento di richiesta di login, lato client vengono controllati la lunghezza di nickname e password e se i due campi sono vuoti. Nel caso in cui la lunghezza di uno dei due campi (o di entrambi) superi i 20 caratteri oppure uno dei due campi (o entrambi) è vuoto viene restituito, a pop-up, un errore di "`username e/o password errati`" (come nella figura in alto a destra). Se questo controllo va a buon fine il client si connette con il server e manda la richiesta di login usano come nickname e password quelli inseriti dall'utente. A questo punto il server controlla se l'utente esiste, se la password è errata e se l'utente è già loggato. In questi casi il server restituisce al client errori del tipo `"Utente inesistente: registrarsi"` nel primo caso, `"Password errata"` nel secondo caso, `"Utente già loggato"` nel terzo caso, e chiude la comunicazione con il client. Nel caso di altri errori durante la connessione o la comunicazione con il server al client viene restituito un generico messaggio di errore.

Tutti i messaggi di errore sono mostrati a pop-up in queste e nelle successive interfacce.

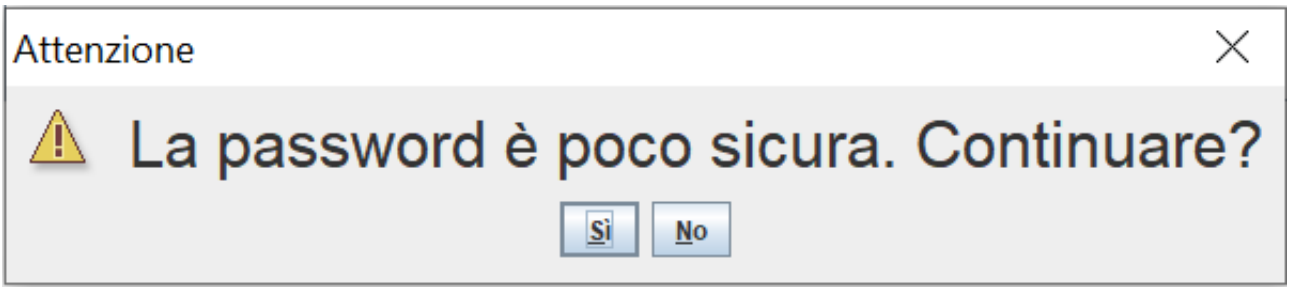
Nel caso in cui il server stabilisce che l'utente non rientra in uno dei casi elencati sopra e può connettersi, restituisce al client un codice che ne indica l'avvenuto login e gli comunica il punteggio al momento del login.

A questo punto il client carica la schermata successiva che gli permette di utilizzare il gioco e può vedere il suo punteggio comunicatogli precedentemente dal server.

I messaggi di errore restituiti all'utente in caso di registrazione sono diversi da quelli dell'operazione di login in quanto si suppone che l'utente non sappia utilizzare il gioco e quindi si cercano di fornire informazioni in più.

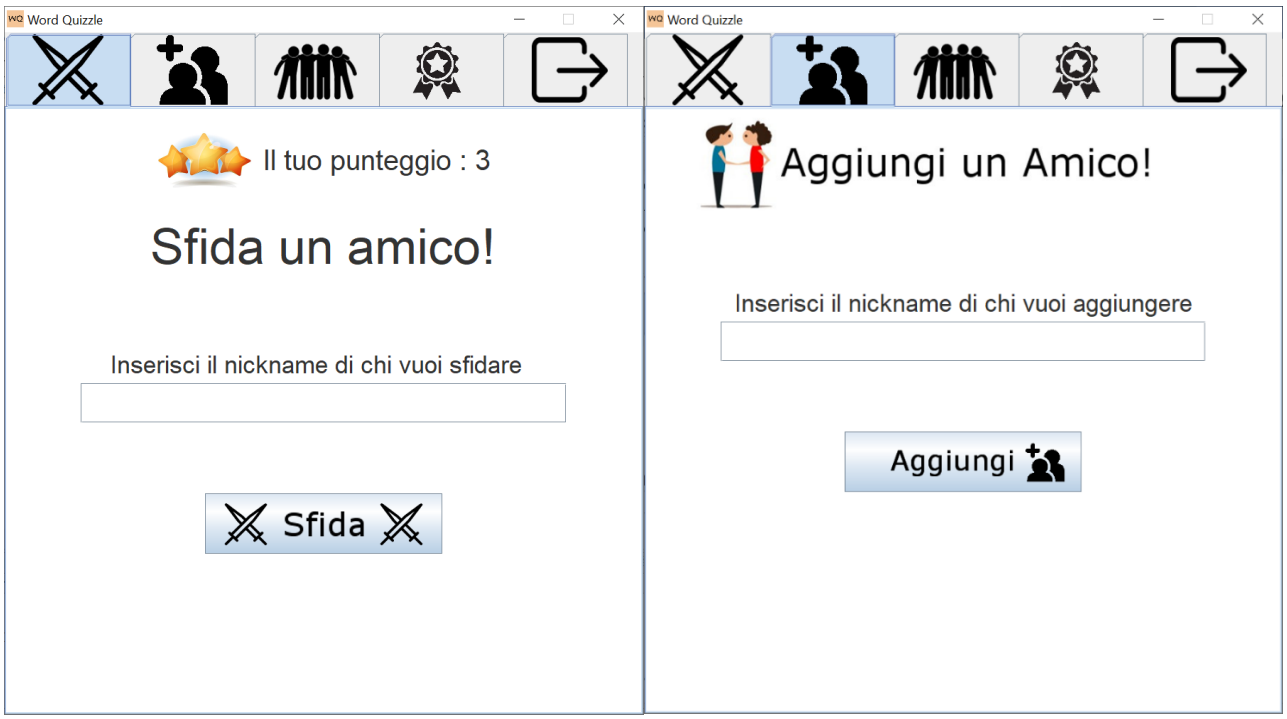
Al momento di richiesta di registrazione, lato client vengono controllati la lunghezza di nickname e password e se i due campi sono vuoti. Nel primo caso viene restituito l'errore `"I campi possono avere massimo 20 caratteri"`, mentre nel secondo caso viene restituito `"username e password non possono essere vuoti"`.

Un ulteriore controllo che viene fatto lato client è quello sulla lunghezza della password nel caso in cui questa non sia vuota o non sia più lunga di 20 caratteri. Nel caso in cui la password sia più corta di 6 caratteri viene considerata pogo sicura e quindi viene richiesta conferma all'utente tramite un messaggio come nella della figura seguente.



In caso in cui l'utente risponda "No" nessuna operazione viene eseguita in attesa di una successiva richiesta di operazione da parte dell'utente.

Nel caso in cui l'utente risponda "Sì" oppure nel caso in cui questa conferma non risulti necessaria perché la password inserita non è più corta di 6 caratteri il client tenta di registrare l'utente connettendosi all'oggetto remoto nel server. Nel caso in cui il nickname inserito dall'utente sia già stato preso da qualcun altro viene restituito all'utente il messaggio `"Username già preso"` invitando quindi l'utente a sceglierne uno nuovo. Se questo non si verifica l'utente si registra e automaticamente viene fatto il login nello stesso modo indicato sopra. Se nell'operazione di registrazione o di login successivi a quest'ultima dovessero verificarsi errori all'utente viene mostrato un messaggio di errore generico o un messaggio di errore più specifico in alcune circostanze (`"Connessione rifiutata"`, `"Registrazione effettuata. Errore durante il login"`).



In caso di login eseguito con successo l'utente visualizza la schermata principale (la schermata in alto a sinistra) dove gli viene mostrato il suo punteggio e dove può fare richieste di sfida. È stata scelta questa come schermata principale, tra le 5 possibili visualizzabili dopo il login, in quanto è la funzionalità principale dell'applicazione.

Per il punteggio l'utente non ha un'operazione specifica che può essere eseguita per vederlo o aggiornato, ma questo viene gestito in modo automatico ed è sempre aggiornato. Viene letto quando un utente si connette e viene aggiornato dopo ogni sfida visto che è l'unico momento in cui questo può cambiare. In

questo modo il punteggio aggiornato è sempre disponibile e si evita che l'utente faccia richieste di aggiornamento anche quando non sono necessarie.

Dalla schermata di sfida quindi un utente può sfidarne un altro al patto che sia un suo amico. Anche in caso di sfida vengono fatti dei controlli lato client. Se l'utente fa richiesta di sfida (cliccando sul pulsante "Sfida") non inserendo nessun nickAmico viene restituito l'errore `"Nickname amico non inserito"`, se il nickAmico inserito è lungo più di 20 caratteri viene restituito `"Nickname amico non valido"`, mentre se il nickAmico è uguale al nickname dell'utente che sta facendo la richiesta viene restituito `"Non puoi sfidare te stesso"`.

Nel caso un cui questi controlli restituiscano un risultato positivo viene inoltrata la richiesta di sfida al server. Il server controlla se esiste un utente con il nick specificato e se è presente tra gli amici di chi fa la richiesta. In caso negativo restituisce rispettivamente `"Utente inesistente"` e `"Tu e` *nickAmico* `non siete amici"`. Se l'utente amico è impegnato in un'altra sfida, non è connesso, scade il tempo massimo di attesa di una risposta oppure ottengo un errore considero la sfida rifiutata e lo segnalo all'utente.

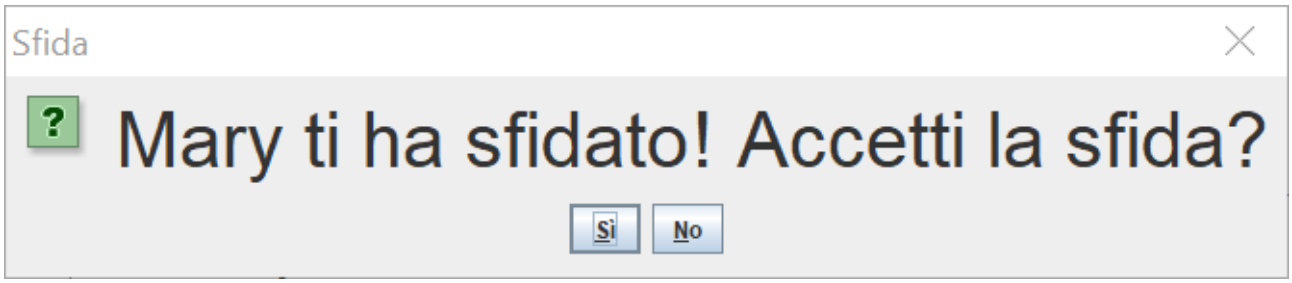
Considero rifiutata la richiesta in caso in cui l'amico è impegnato in un'altra sfida o se non è connesso in quanto è poco probabile che l'utente risponda alla richiesta prima dello scadere del tempo. In questo modo l'utente non deve aspettare lo scadere del tempo di attesa di risposta prima di poter fare un'altra richiesta (la gestione delle richieste UDP è stata implementata in accordo con queste supposizioni).

Nel caso in cui la richiesta vada a buon fine viene caricata la schermata della sfida vera e propria (descritta in seguito).

L'accettazione della sfida può essere fatta in una qualsiasi delle 5 schermate che l'utente può visualizzare dopo il login ma fuori da una sfida.

L'accettazione prevede la comparsa di un messaggio pop-up che richiede una conferma o un rifiuto da parte dell'utente sfidato mostrando chi è l'utente che sfida.

Un esempio di pop-up di accettazione è quello della figura in basso dove viene chiesto all'utente sfidato se vuole accettare la sfida lanciata dall'utente con il nickname "Mary".



In caso di "No" o nel caso in cui la finestra venga chiusa tramite la X in alto a destra la sfida viene rifiutata.

Se l'utente sceglie "Si", cioè sceglie di accettare a sfida, se accetta prima della scadenza del tempo massimo dell'accettazione allora inizia la sfida e viene caricata la schermata di sfida vera e propria, altrimenti l'utente riceve un messaggio di `"Richiesta scaduta"`.

Se si verifica un errore durante l'accettazione viene segnalato all'utente e la sfida viene annullata e l'utente può proseguire normalmente.

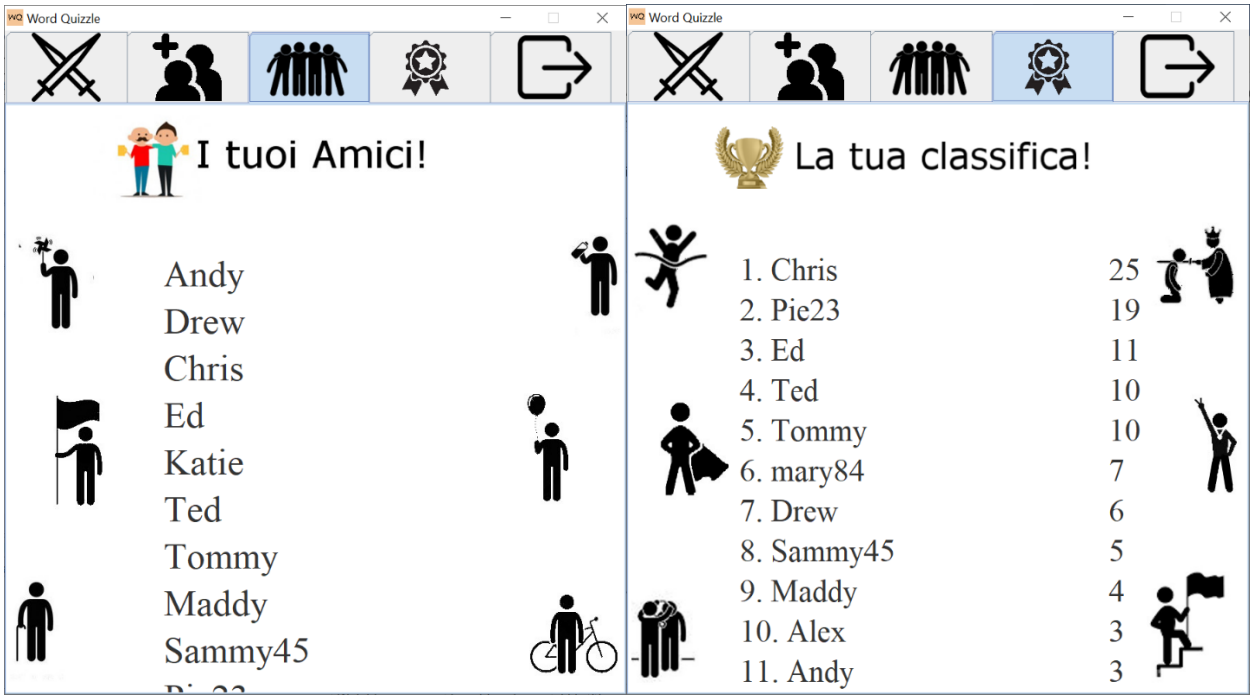
Un’altra operazione permessa all’utente è l’aggiunta di un amico che può essere richiesta tramite la schermata nella seconda figura in alto a destra. Un amico può essere aggiunto inserendo il suo nickname nello spazio apposito e poi cliccando sul pulsante “Aggiungi”.

Come per la sfida, vengono fatti i controlli lato client sul nick inserito controllando se è non stato inserito, se è più lungo di 20 caratteri e se si sta tentando di aggiungere se stessi come amico sollevando gli errori `"Nickname amico non inserito"`, `"Nickname amico non valido"` e `"Non puoi aggiungerti come amico"`.

Se il nickAmico inserito è valido viene mandata la richiesta al server che controlla se esiste un utente con quel nick e se si sta cercando di aggiungere qualcuno che è già amico. Se non esiste o se sono già amici vengono mostrati rispettivamente gli errori `"Utente inesistente"` e `"Siete già amici"`.

Altrimenti il server crea l’amicizia facendo diventare l’utente amico di nickAmico e nickAmico amico dell’utente (l’amicizia è bidirezionale) e lo comunica all’utente (`"Ora tu e` *nickAmico* `siete amici!"`).

Come nel caso di login o registrazione, gli spazi vuoti nel nickAmico inserito non vengono considerati e vengono rimossi in maniera trasparente all’utente.



Un utente può far richiesta di vedere la lista con i propri amici cliccando sull’immagine con le 4 persone (quella evidenziata nella barra in alto nell’immagine in alto a sinistra).

In questo modo viene mostrato l’elenco con i propri amici (figura in alto a sinistra) oppure se non se ne hanno viene mostrata la scritta `"Nessun amico da mostrare. Aggiungi Qualcuno!"`.

Un utente può far richiesta di vedere la classifica con sé stesso e i propri amici cliccando sull’immagine con la medaglia (quella evidenziata nella barra in alto nell’immagine in alto a destra).

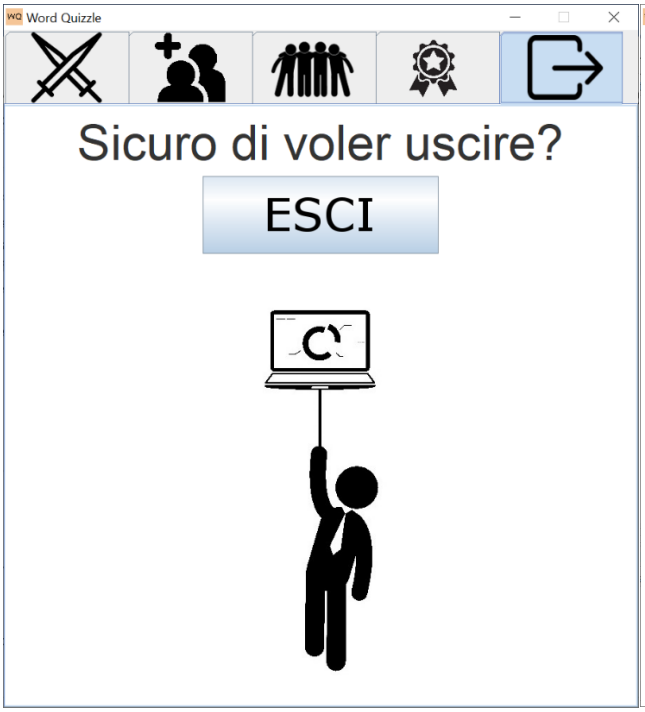
In questo modo viene mostrata la posizione propria e dei propri amici all’interno della classifica, il nome e il punteggio (figura in alto a destra). La classifica è ordinata per punteggio decrescente. La classifica ha

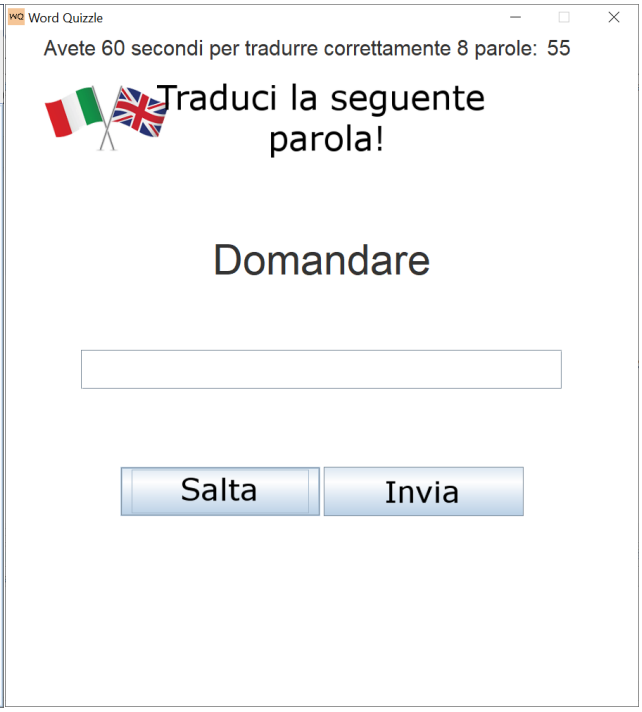
sempre almeno una riga in quanto è compreso l'utente che ne fa richiesta il quale esiste sicuramente (ad esempio se Alex fa richiesta di vedere la propria classifica Alex comparirà sempre nella classifica quindi, anche se potrebbe non avere amici, la classifica ha sempre almeno una riga, cioè quella con il punteggio di Alex).

Se la lista di amici o la classifica non entrano nella schermata è possibile lo scroll per poter visualizzare i nomi che non ci sono entrati.

È stato scelto, tra le varie alternative, di mandare la richiesta di aggiornamento e visualizzazione della lista di amici o di classifica quando l'utente clicca sul tab che mostra la lista o la classifica. Questo perché permetterne la richiesta quando si trova su un'altra schermata, come ad esempio quella della sfida o dell'aggiunta di un amico, potrebbe far si che si vengano a generare richieste inutili in quanto da quelle schermate i risultati della richiesta comunque non sarebbero visualizzabili e l'utente potrebbe fare richiesta di aggiornamento e poi passare a una successiva attività. Invece aggiornando la lista di amici o la classifica solo quando l'utente fa richiesta di vederla evita aggiornamenti inutili facendo fare anche meno lavoro al server il quale non deve risponder a richieste inutili. In più evitando un aggiornamento su richiesta dell'utente si evita la possibilità di visualizzare un risultato non aggiornato. Ad esempio un utente potrebbe richiedere di aggiornare la propria lista di amici, ma non visualizzarla. Successivamente potrebbe aggiungere altri amici e visualizzare la lista senza averla aggiornata ulteriormente. In questo caso l'utente vedrebbe una lista parziale e non completa dei suoi amici.

La scelta fatta evita il verificarsi dell'ultimo evento e cerca di ridurre il numero di aggiornamenti. Se un utente sta visualizzando la sua lista di amici (o la sua classifica) e viene aggiunto come amico da qualcun altro quest'ultimo non compare nella lista di amici (o nella classifica) fino all'accesso successivo alla lista (o alla classifica). Questa soluzione è stata preferita all'altra che consisteva nel fare richieste periodiche al server mentre l'utente visualizzava la propria lista (o classifica) così da cercare di evitare un sovraccarico di lavoro da parte del server, considerato che il numero di client connessi contemporaneamente al server può essere elevato.





Per uscire dal gioco ci sono 2 modi: andare sulla schermata di logout (figura in alto a sinistra) cliccando sulla 5 figura nella barra in alto presente in ognuna delle 5 schermate descritte sopra oppure cliccando sulla croce in alto a destra.

Con il primo modo viene mostrata la schermata presente nell'immagine in alto, dove si chiede una confera all'utente prima di uscire che viene data cliccando sul pulsante esci. Nel secondo modo questa conferma non è prevista e quindi chiude direttamente l'applicazione. Entrambe le chiusure sono state gestite e comunicano al server la volontà di logout prima di chiudere il gioco e non richiedono una risposta da quest'ultimo.

Il secondo metodo di chiusura quindi può essere usato dall'utente in qualsiasi delle 5 schermate. L'utente non deve obbligatoriamente essere sulla schermata di login per poter uscire ma potrebbe ad esempio uscire mentre si trova nella schermata di sfida.

Si può passare da una schermata ad un'altra semplicemente cliccando sulla relativa immagine nella barra in alto della schermata in cui si ci trova (sulle spade per tornare alla schermata di sfida, sulle 2 persone con il + per aggiungere un amico, ecc..)

In tutte le richieste fatte al server in tutte le 5 schermate, in caso di errore di comunicazione con il server, viene segnalato l'errore all' utente e viene chiusa l'applicazione.

Dopo l'accettazione di una sfida andata a buon fine inizia la sfida vera e propria e all'utente viene aperta la schermata con la prima parola tra quelle da tradurre (come nella figura in alto a destra). Durante la sfida viene indicato il numero di parole totali da tradurre, il tempo a disposizione e il tempo rimanente (tutto nella riga in alto. Il tempo rimanente è il numero che si aggiorna nella riga più in alto dopo i ":" (nella figura in alto a destra è 55)).

Viene poi indicata la parola e sono messe a disposizione dell'utente tre possibili operazioni: passare alla parola successiva senza dare una risposta con il pulsante "Salta", inviare la traduzione e passare alla parola successiva con il pulsante "Invia" oppure uscire dalla sfida con la croce (X) in alto a destra.

Successivamente userò il termine tradurre anche per le parole saltate facendo la distinzione tra i due casi solo dove necessario.

La sfida finisce quando tutti e due gli sfidanti traducono tutte le parole, quando il tempo scade oppure quando si abbandona la sfida (L'abbandono viene considerato una resa da parte dell'utente).

Se entrambi gli sfidanti finiscono le parole ognuno ottiene il punteggio dell'altro e vede se ha vinto, perso o pareggiato e la sfida termina prima dello scadere del tempo. Se invece almeno uno dei due non traduce tutte le parole in tempo viene interrotto dal timer. E a questo punto viene calcolato il punteggio e si procede come sopra.

Al termine della sfida viene indicato all'utente se ha vinto, perso o pareggiato, i suoi punti totalizzati durante la sfida, quelli dell'avversario e i punti extra guadagnati (se guadagnati). Dopo il termine di una sfida terminata normalmente (cioè senza abbandono) il punteggio dell'utente viene aggiornato.

L'utente che finisce prima deve aspettare la terminazione della sfida dell'altro.

Nel caso in cui solo uno dei due sfidanti abbandona la sfida l'altro continua normalmente ma alla fine risulterà vincitore per abbandono da parte dell'altro sfidante. Colui che ha abbandonato non deve aspettare che anche l'altro finisca prima di poter tornare alla schermata principale e fare altre richieste.

Se entrambi abbandonano nessuno vince e la sfida si conclude nel momento in cui anche il secondo esce, ma anche in questo caso il primo non deve aspettare che abbandoni anche il secondo prima di tornare alla schermata principale.

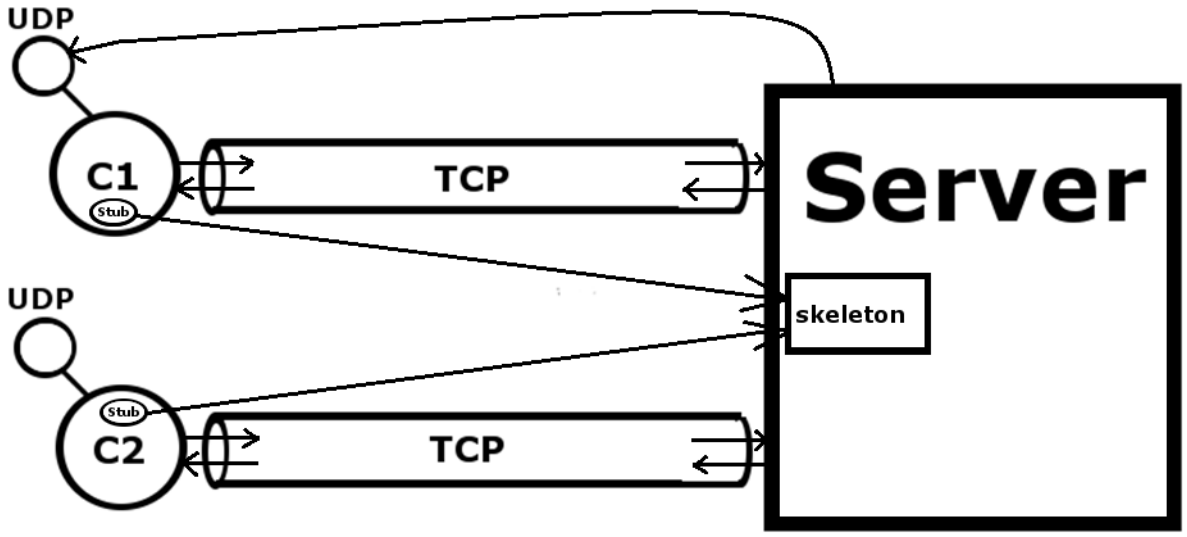
In caso di abbandono della sfida, il punteggio dell'utente non viene modificato, cioè l'utente non guadagna né perde punti.

Durante una sfida un utente non può chiudere il gioco (da interfaccia grafica. Però può farlo terminando il processo e ciò è stato gestito).

Sono stati lasciati, per alcuni errori, ulteriori informazioni stampate a riga di comando per fare verifiche durante il testing.

Comunicazione tra client e server

Schema semplificato delle comunicazioni tra client e server (due client C1 e C2).



La comunicazione tra client e server è stata realizzata seguendo le linee guida fornite.

Un client quando viene creato si mette in ascolto su una porta dove riceve i pacchetti UDP. Qui si limiterà a ricevere e riceverà solo richieste di sfida da parte di utenti registrati come amici. La porta su cui un client riceve le richieste di sfida viene comunicata al server al momento del login.

Il client, quindi, non riceve mai altre informazioni o richieste tramite UDP e non risponde mai tramite UDP ma solo tramite TCP.

Viene usata la comunicazione UDP solo per mandare le richieste di sfida al client. Ad esempio, se un client C2 vuole sfidare un client C1, C2 comunica al server la volontà di sfida tramite la connessione TCP instaurata precedentemente e il server manda, tramite UDP, la richiesta di sfida a C1. C1 a questo punto risponde tramite la connessione TCP che ha instaurato precedentemente con il server.

Il messaggio UDP mandato come richiesta di sfida contiene come dati solo il nick dell'amico che sta sfidando.

È stato scelto di non usare una porta fissa per ricevere i pacchetti UDP così da evitare problemi nel caso in cui quella porta sia già utilizzata. È il sistema che sceglie una porta non utilizzata la quale viene comunicata dal client al server al momento del login.

Per rispondere alle richieste utilizzerà la connessione TCP che viene creata quando un utente fa richiesta di login.

Se la richiesta di login non fallisce, il client utilizzerà questa connessione per tutte le altre richieste (l'utilizzo della connessione TCP è descritto nel successivo paragrafo "Protocollo di comunicazione"), mentre se fallisce, la connessione viene chiusa.

La registrazione di un utente invece è stata implementata mediante RMI (come da specifiche), quindi, quando un utente fa richiesta di registrazione, il client invoca il metodo dell'oggetto remoto per aggiungere un utente. Questo metodo ritorna un codice che indica se la registrazione è andata a buon fine oppure se per qualche motivo è fallita.

L'RMI è stato usato solo per la registrazione di un utente quindi mette a disposizione del client un solo metodo, quello per registrarsi.

Nella fase di registrazione, se questa va a buon fine è previsto un login automatico con le credenziali appena registrate, e viene creata la connessione TCP dove inviare la richiesta di login al server.

Quindi tutta la comunicazione tra client e server, escluso il messaggio di sfida ricevuta e la registrazione di un utente, avviene tramite la connessione TCP.

Protocollo di comunicazione TCP

È stato definito un protocollo di comunicazione tra client e server. Ogni messaggio che arriva al server è indipendente da quello precedente, cioè non vengono fatti controlli sulla validità del messaggio. Ad esempio se arriva al server un messaggio per passare alla parola successiva all'interno della sfida, non viene fatto un controllo sui messaggi precedenti per verificare che il client sia davvero in una sfida oppure no. Questo perché i messaggi possibili sono stati limitati tramite l'interfaccia grafica del client. Cioè ad esempio, se un client non è all'interno di una sfida, un utente non visualizzerà mai il pulsante per passare alla parola successiva ed è quindi impossibilitato a mandare questo tipo di messaggio al server. Questo concetto è stato usato anche per limitare tutti gli altri messaggi possibili in base a cosa sta facendo l'utente.

Messaggi che il client manda al server.

1- login
   che è seguito dai parametri *username*, *password* e *portUDP*, dove username è il nickname di chi sta chiedendo di connettersi, password è la password associata al nick e portUDP è la porta su cui il client si mette in attesa di richieste di sfida.

Questo messaggio è utilizzato per eseguire il login dell'utente *username.*

2- getScore
   Questo messaggio è utilizzato per ottenere dal server il punteggio dell'utente.
3- aggiungiamico
   che è seguito dal parametro *nickAmico* che indica il nick dell'utente che si vuole aggiungere come amico.
   Questo messaggio è utilizzato per aggiungere l'utente *nickAmico* come amico.
4- mostraAmici
   Questo messaggio è utilizzato per richiedere la propria lista di amici.
5- mostraClassifica
   Questo messaggio è utilizzato per richiedere la propria classifica.
6- logout
   Questo messaggio è utilizzato per eseguire il logout, quindi per disconnettere il client.
7- sfida
   che è seguito dal parametro *nickAmico* che indica il nick dell'utente che si vuole sfidare.
   Questo messaggio è utilizzato per sfidare *nickAmico.*
8- notconfirm
   Questo messaggio è utilizzato per rifiutare una richiesta di sfida.
9- confirm
   Questo messaggio è utilizzato per accettare una richiesta di sfida.
10- saltaparola
   Questo messaggio è utilizzato all'interno di una sfida per passare alla parola successiva senza inviare la traduzione della parola corrente.
11- inviaparola
   che è seguito dal parametro *parola* che indica la parola inserita dall'utente come traduzione alla parola datagli da tradurre.
   Questo messaggio è utilizzato all'interno di una sfida per passare alla parola successiva inviando la traduzione della parola corrente.
12- scaduto
   Questo messaggio è utilizzato all'interno di una sfida per indicare al server che il tempo a disposizione per la traduzione delle parole è scaduto.
13- uscitabattaglia
   Questo messaggio è utilizzato all'interno di una sfida per indicare la volontà di abbandono della sfida da parte dell'utente e del ritorno alla schermata principale.

Messaggi di risposta del server

1- risposta a login
   i. ok: se il login è andato a buon fine
   ii. already: se il login ha fallito perché l'utente è già loggato
   iii. notexist: se non esiste un utente con il nick indicato
   iv. errpw: se la password inserita non corrisponde a quella associata al nick indicato
2- risposta a getScore
   i. *n*: dove *n* è il punteggio dell'utente
3- risposta ad aggiungiamico

i. 0: se la relazione di amicizia è stata inserita con successo
ii. 1: se la relazione di amicizia è già presente
iii. -1: se l'utente da aggiungere come amico non esiste

4- risposta a mostraAmici
   i. oggetto JSON con la forma {"Friends": "[lista di amici]"} dove la lista di amici può essere vuota
5- risposta a mostraClassifica
   i. array JSON contenente oggetti con la forma {"nick":"Alex","score":3,"position":9} (dati di esempio), dove "score" indica il punteggio dell'utente "nick" e "position" la sua posizione in classifica.
6- risposta a logout
   Non è prevista una risposta del server al client
7- risposta a sfida
   i. -1: se l'utente sfidato non esiste
   ii. 0: se la richiesta e stata accettata e non ci sono stati errori
      Successivamente in server invia al client il tempo a disposizione per la sfida, le informazioni per la sfida stessa (numero di parole totali) e la prima parola da tradurre.
   iii. 1: se l'utente sfidato non è presente nella lasta di amici
   iv. 2: se la richiesta è stata rifiutata o se ci sono stati errori
8- risposta a notconfirm
   Non è prevista una risposta del server al client
9- risposta a confirm
   i. 0: se la rconferma di sfida è andata a buon fine
      Successivamente in server invia al client il tempo a disposizione per la sfida, le informazioni per la sfida stessa (numero di parole totali) e la prima parola da tradurre.
   ii. 1: se la richiesta è stata accettata fuori tempo massimo
   iii. 2: se ci sono stati errori
10- risposta a saltaparola
   i. parola successiva se non sono terminate
   ii. "end" se sono terminate. Successivamente il server invia al client il risultato della sfida (pareggio, vincita, …)
11- risposta a inviaparola
   i. parola successiva se non sono terminate
   ii. "end" se sono terminate. Successivamente il server invia al client il risultato della sfida (pareggio, vincita, …)
12- risposta a scaduto
   i. risultato della sfida (pareggio, vincita, …)
13- risposta a uscita battaglia
   Non è prevista una risposta del server al client

Server: Gestione dei file e delle traduzioni dal servizio esterno

È stato scelto di implementare un server multithreaded invece che fare multiplexing dei canali mediante NIO (la stessa decisione è stata presa per il client). La descrizione dei thread attivati è rimandata al capitolo successivo.

È stato scelto di leggere ogni file necessario una volta sola, quando il server viene avviato ma prima di permettere connessioni con i client. Questo perché le scritture/letture da file sono molto costose e dato che le scritture sono necessarie per rendere persistenti i dati aggiornati si è cercato di limitare il numero di letture.

Quindi per ogni file letto viene creata una struttura dati che ne memorizza il contenuto e il server lavora su queste. Successivamente questi dati aggiornati verranno scritti (quindi i dati all'interno di queste strutture dati sono sempre aggiornati). Questa scelta è nata dalla supposizione che il server abbia una quantità di memoria elevata.

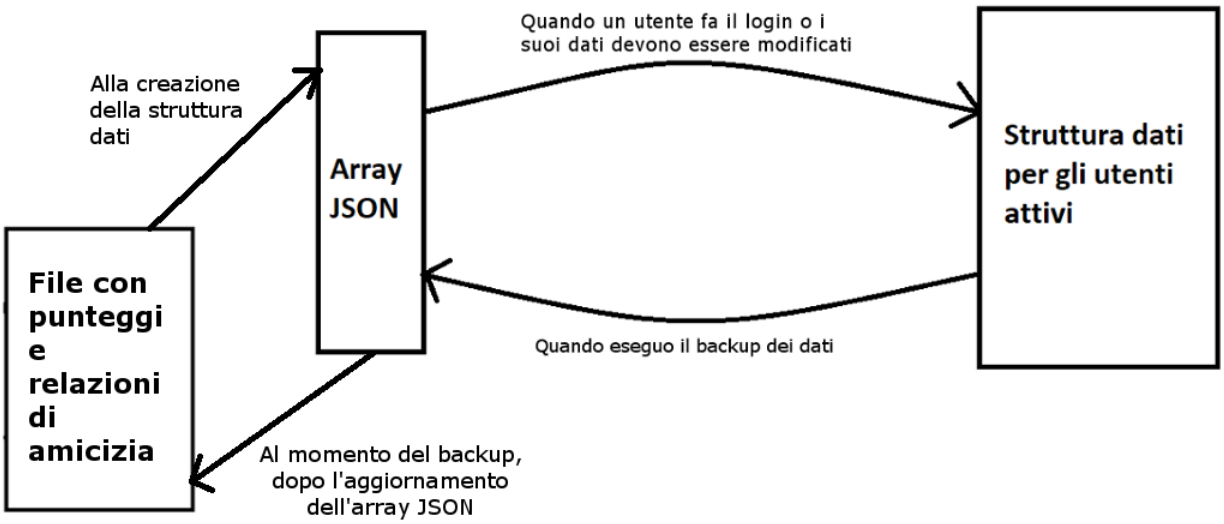
I file sono 3 e la loro gestione è diversa, come il loro scopo.

Il primo è Dizionario.txt. Questo file contiene il dizionario. Si suppone che questo file non abbia mai bisogno di essere modificato quando il server è in esecuzione. Il server lo legge all'inizio, crea un array di parole con le parole al suo interno e le conta per poi verificare se questo numero è significativo. Dopo la lettura e la creazione dell'array questo file non viene più utilizzato quindi viene chiuso. Visto che non viene mai modificato non è mai necessario un suo aggiornamento, perciò non verrà più aperto fino alla prossima esecuzione del server.

Il secondo file è Utenti.json. Questo file contiene oggetti JSON formati da nick e password. Viene usato solo per le registrazioni e i login degli utenti e non contiene altri dati riguardo questi. Viene letto solo all'inizio e i suoi dati vengono salvati in un array JSON sul server. Successivamente questo file verrà solo scritto. Al momento del login o della registrazione il controllo di esistenza dell'utente o di correttezza della password viene fatto sulla struttura dati creata con i dati del file.

Questa struttura dati viene mantenuta sempre aggiornata. Al momento di una registrazione da parte di un utente andata a buon fine, si è scelto di salvare subito i nuovi dati sul file. I dati sui nuovi utenti (nickname e password) vengono visti come informazioni sensibili e una perdita di molte informazioni di questo tipo viene vista come non tollerabile. In più si suppone che le operazioni di registrazioni siano molto meno frequenti rispetto alla altre. In questo modo, anche se dovesse esserci un crash da parte del server, le informazioni di questo tipo che possono essere perse sarebbero poche.

Il terzo e ultimo file usato dal programma è Amicizie.json. Questo file contiene un array di oggetti JSON formati da nick, score e friends, e ne memorizza, quindi per l'utente con un certo nickname lo score e gli amici. Anche questo file viene letto solo all'inizio e i suoi dati vengono salvati in un array JSON. Questa struttura dati non è sempre aggiornata. Il server utilizza una struttura dati per gli utenti attivi o comunque per gli utenti che hanno subito aggiornamenti ai loro dati (score o amici). Quando si ha bisogno di fare una modifica ai dati di un utente, l'utente viene "caricato" dall'array che mantiene i dati letti al file a quest'altra struttura dati. Ogni tot tempo o quando l'utente che ha accesso al server fa richiesta di backup, i dati in questa struttura dati vengono usare per aggiornare l'array JSON e rimossi dalla struttura dati se l'utente non è attivo. Dopo i dati nell'array JSON vengono memorizzati su file.



È stata creata questa struttura dati perché mantiene informazioni in più rispetto a quelle contenute nell'array JSON e per le informazioni "comuni" come il punteggio è una struttura dati più veloce da modificare invece che dover ricreare l'oggetto JSON ad ogni modifica.

È stato scelto di fare il backup ogni 5 minuti (è stato scelto un intervallo di tempo abbastanza breve a scopo di testing).

Quando l'utente fa richiesta di terminazione del server, non vengono più permesse registrazioni e, dopo che tutti i client si sono disconnessi, anche se il tempo prima del prossimo backup non è scaduto, si fa un ultimo backup straordinario prima di chiudere il server così da assicurarsi che le informazioni memorizzate siano aggiornate.

Riguardo le traduzioni ottenute dal servizio esterno, queste sono richieste, come da specifica, prima dell'inizio della partita ma dopo che è stata ricevuta l'accettazione della sfida. Per velocizzare la generazione delle parole con le relative traduzioni da parte del server, la traduzione di una parola viene memorizzata in una struttura dati non solo per il tempo della sfida ma anche per quelle successive. Quindi se una certa parola non è stata mai tradotta vengono chieste le possibili traduzioni al servizio esterno, mentre se è già stata tradotta in passato si usano le traduzioni richieste precedentemente.

Thread e strutture dati utilizzate

Client

Il client è stato realizzato multithreaded.

Il primo thread mandato in esecuzione all'avvio del client è quello del main. Da questo viene creato in modo esplicito un thread che esegue il metodo run della classe AskBattle. Questo thread si mette in attesa di ricevere pacchetti UDP per le richieste di sfida. Quando viene ricevuta una richiesta di sfida questo nuovo thread ne crea un altro che che esegue il metodo run della classe Concurrent. Questo thread si occupa di

richiedere l'accettazione o il rifiuto della sfida dal thread sfidato. Posso avere contemporaneamente al massimo un solo Thread che esegue il metodo della classe Concurrent. Questo perché a un client non può arrivare una richiesta di sfida se ne ha già una in sospeso. Il thread termina con il rifiuto della sfida o con la sua accettazione. Con il rifiuto (o con un errore) termina il proprio lavoro e successivamente la sua esecuzione, mentre con l'accettazione viene terminato dopo la sfida. Anche il thread che inizialmente mostra la schermata all'utente, con la creazione della nuova schermata da parte del thread Concurrent, viene terminato. La libreria awt, usata per la gestione degli eventi, crea dei suoi thread in modo non controllato dal programmatore, ma terminando quelli, invece, creati in modo esplicito da esso.

I thread non usano in modo concorrente nessuna struttura dati.

Nel client non vengono usate particolari strutture dati, ma perlopiù stringhe e interi.

Threads del server

Il server è stato realizzato multithreaded.

Anche per il server ci sono dei thread creati in automatico. Il server implementa la registrazione mediante RMI. Ciò crea un thread per ogni connessione del client.

Successivamente il server crea un thread che si occupa di fare un backup periodico dei dati riguardanti punteggio e relazioni di amicizia degli utenti e un thread che gestisce l'interazione dell'utente con il server, leggendo i comandi dati da linea di comando e occupandosi della loro esecuzione (svegliando altri thread, oppure eseguendo direttamente il necessario).

Poi il server crea il pool di thread che si occuperà di rispondere alle richieste del client. È stato usato un pool che non ha limiti sulla dimensione.

Quindi i thread del server sono:

- uno che è quello del main il quale accetta le connessioni con i client;
- quelli creati in automatico con l'RMI;
- uno per il backup periodico;
- uno per la gestione dell'interazione con l'utente tramite linea di comando;
- quelli del pool (uno per client).

Questi threads (escluso quello del main) sono attivati al partire dal thread main.

Concorrenza e strutture dati all'interno del server

Le strutture dati dove sarebbero potuti sorgere problemi dovuti alla concorrenza sono:

- Il file che contiene i dati delle registrazioni degli utenti e il JSONArray a lui associato.
  La lettura di questo file non è soggetta a nessun tipo di concorrenza in quanto viene letto solo una volta, al momento della creazione dell'oggetto che mette a disposizione metodi per la gestione, quando ancora nessun thread che ha la possibilità di scrivervi è stato creato.
  Invece, riguardo la scrittura su file, potrebbero essere attivi più thread contemporaneamente attraverso l'invocazione del metodo dell'oggetto remoto.
  Per questo, la registrazione di un utente è stata implementata utilizzando lock per scrittori e lettori.

Questa distinzione tra lettori e scrittori è stata fatta, non per la scrittura sul file, ma per la gestione dell'array JSON.
Quando un client si registra con dei dati validi aggiunge un oggetto JSON all'array e poi riporta su file i dati aggiornati. Questo array JSON però viene letto anche da altri threads (quelli che si occupano di rispondere alle richieste dei client). Quindi durante la registrazione il thread acquisisce la lock in scrittura, visto che nessun'altro thread deve poter leggere o scrivere l'array o il file, mentre quando un thread legge l'array acquisisce la lock in lettura che permette anche la lettura di più threads contemporaneamente.
Quindi i threads che possono usare queste strutture dati in modo concorrente sono i threads generati con l'RMI e i threads del pool.

- Il file che mantiene punteggio e amicizie dell'utente. Su questo file non c'è concorrenza in quanto viene letto solo all'inizio dal thead main e successivamente viene solo scritto dal thread che si occupa del backup. La struttura dati a lui associata, invece è soggetta a concorrenza, e lo stesso per la struttura dati che mantiene le modifiche tra un backup e il successivo e le informazioni sugli utenti attivi. Queste 2 strutture dati vengono usate insieme e prevedono, come per il file che mantiene i dati delle registrazioni degli utenti, la lock in scrittura e la lock in lettura. La lock in scrittura viene presa solo dal thread che fa il backup, mentre quella in lettura dai threads Worker (che rispondono alle richieste dell'utente). Le strutture dati uilizzate sono un array JSON e una ConcurrentHashMap che ha come oggetti oggetti di tipo User. Negli oggetti di tipo User la lista degli amici è stata implementata usando un Vector di stringhe.
- La struttura dai che contiene le traduzioni delle parole in italiano, recuperate tramite una chiamata HTTP GET al servizio esterno. questa struttura dati può essere letta e scritta in modo concorrente. Per risolvere questo problema è stata creata, al posto di una HashMap, una ConcurrentHashMap. L'unica operazione di modifica è l'inserimento di nuove coppie <key,value> che si può facilmente realizzare, non avendo problemi dovuti alla concorrenza, con l'operazione atomica putIfAbsent. Le uniche operazioni di lettura riguardano le chiavi e i valori ad esse associate. Quindi, non avendo la necessità di utilizzare una lock per l'intera struttura dati (considerate le operazioni previste su di essa), è stato scelto di usare questa struttura dati (che prevede il lock striping) così da avere il vantaggio dell'aumento del parallelismo e della scalabilità.
- La ConcurrentHashMap usata per gestire le sfide. Questa contiene tutte le sfide in attesa di essere accettate o in corso. Le sfide dei client vengono gestite separatamente, ogni thread gestisce la sfida del client che è incaricato di servire. I threads dei due client coinvolti in una sfida però hanno bisogno di comunicare prima dell'inizio della sfida, per verificare se questa è stata accettata/rifiutata e per ottenere le parole che devono essere uguali per entrambi, e alla fine, per comunicare al client la vittoria/il pareggio/ la sconfitta.
  È stata definita uno specifico procedimento per queste comunicazioni
  - Comunicazione precedente a una sfida.
    Chiamiamo il thread sfidante T1 e il thread sfidato T2.
    T1 riceve la richiesta da parte del client di volerne sfidare un altro. Allora T1 manda la richiesta di sfida con un pacchetto UDP al client sfidato.
    Questo comunica la sua risposta al thread T2. Se è un rifiuto, T2 sveglia T1 e si mette in ascolto della prossima richiesta. Se all'arrivo del segnale di T2 T1 è già sveglio, T1 ignora l'azione di T2, cancella la sfida dalla struttura dati che le contiene tutte e comunica al client il rifiuto. Se invece T1 non era sveglio allora si risveglia, cancella la richiesta di sfida e comunica al client il rifiuto della sfida.

Se la risposta è un'accettazione di sfida allora T2 setta la battaglia come accettata sveglia T1. Se l'accettazione arriva oltre il tempo massimo allora viene restituito un codice apposito a T2 che comunicherà la cosa al client.
Se l'accettazione viene fatta in tempo allora T1, se le parole non sono state ancora generate da T2 si mette in attesa si esse. Se durante la generazione si verifica un errore viene comunicato al client che la sfida è stata rifiutata. Altrimenti T1 legge le parole e inizia la gestione della sfida per il client da lui gestito.
Se l'accettazione viene fatta in tempo da T2, T2 genera le parole della sfida e richiede le traduzioni. Se durante questa fase avviene un errore, T2 setta la sfida come rifiutata (per T1), manda un segnale per svegliare T1 in caso fosse in attesa delle parole per la sfida e cancella la sfida dalla struttura dati comunicando al client che aveva accettato la sfida che c'è stato un errore. Se invece le parole e le traduzioni vengono generate/recuperate correttamente allora T2 legge le parole e inizia la gestione della sfida per il client da lui gestito.

- Comunicazione successiva a una sfida
  Quando un client ha finito di tradurre tutte le parole il thread T1 deve calcolare il risultato e restituirlo al client.
  Quindi, quando le parole finiscono, T1 aggiorna il punteggio del proprio utente nel campo apposito nella struttura dati che rappresenta la sfida. Successivamente controlla se l'altro utente ha abbandonato la sfida. Se si allora cancella la sfida e comunica al client la vittoria calcolandone il relativo punteggio. Se invece l'altro utente non ha abbandonato controlla se l'avversario ha terminato prima controllando se è stato già inserito il suo punteggio. In questo caso so che il thread avversario T2 sta aspettando l'inserimento del punteggio da parte di T1, allora T1 legge il punteggio avversario, sveglia T2 ed esce. Se invece T2 non ha ancora inserito nessun punteggio T1 è il primo ad inserire e quindi deve aspettare l'inserimento del punteggio oppure la conferma di abbandono da parte di T2. Quando T2 inserisce uno dei 2 dati sveglia T1 il quale memorizza il punteggio o l'abbandono e rimuove la sfida. T1 poi calcola il risultato della sfida in base alle informazioni appena acquisite e comunica al client il risultato.
  T2 esegue lo stesso procedimento di T1.
- Comunicazione per l'abbandono di una sfida
  Quando un utente fa richiesta di abbandono, il thread che ne gestisce le richieste T1 controlla se anche l'avversario ha abbandonato. Se si allora la sfida viene cancellata e T1 si mette in attesa delle successive richieste. Altrimenti T1 è il primo a segnalare l'abbandono. Quindi lo segnala e controlla se l'altro thread T2 stava aspettando un inserimento di punteggio da parte di T1. In questo caso T1 sveglia T2 ed esce, altrimenti si limita ad uscire.

Quindi i ruoli dei 2 threads partecipanti a una sfida sono ben definiti, cioè, ad esempio, è sempre il thread "sfidato" che genera le parole, o che in caso di errore durante questa fase cancella la sfida, mentre è sempre il thread "sfidante" che si occupa di cancellare la sfida in caso di rifiuto o in caso di scadenza del tempo massimo di attesa.

Tutte operazioni eseguite da T1 e T2 sulla struttura dati che contiene la sfida vengono eseguite in mutua esclusione all'interno di una lock. Ogni sfida ha la propria lock e le proprie variabili di condizione per far attendere il thread in attesa di un evento e per risvegliarlo al verificarsi di quest'ultimo. Non avrebbe senso usare una lock comune a tutte le sfide visto che ogni coppia di

threads legge e scrive solo su un inserimento, su una riga, della ConcurrentHashMap. Le lock su quest'ultima sono gestite in automatico dalla struttura dati.

Sulle strutture dati non descritte nell'elenco precedente, come ad esempio la struttura dati che rappresenta il dizionario, non ci sono mai problemi legati alla concorrenza. Ad esempio nel caso del dizionario non ci sono problemi perché viene creata la struttura dati all'inizio, quando c'è un solo thread, e solo successivamente vengono messi a disposizione metodi i quali sono solo in lettura e mai in scrittura.

Descrizione delle classi

Le classi definite sono

1. AskBattle.java
2. Battle.java
3. Client.java
4. Concurrent.java
5. Dictionary.java
6. Interfaces.java
7. ManagerBattles.java
8. ManagerFileFriends.java
9. ManagerFileRegistrations.java
10. ManagerServer.java
11. ManagerUsers.java
12. ParametersBattle.java
13. RemoteRegistration.java
14. Server.java
15. Translates.java
16. User.java
17. Worker.java
18. WorkerOnFile.java

Le interfacce definite sono:

1. Registration.java

Classi usate dal client

Le classi usate dal client sono: AskBattle.java, Client.java, Concurrent.java e Interfaces.java

La classe Client.java è quella che contiene il main. Può prendere in input 2 parametri (il path dove trovare le immagini necessarie all'interfaccia grafica e la codifica) oppure nessuno.

Quando viene eseguita crea un'oggetto della classe Interfaces, passando i dati necessari come le informazioni sul server, sulle immagini e l'encoding) e un oggetto della classe AskBattle (comunicandogli il

riferimento dell'oggetto Interfaces e l'encoding) e inizializza i valori che devono essere comuni tra i 2 oggetti appena creati. Successivamente richiama il metodo dell'oggetto della classe Interfaces per creare l'interfaccia grafica iniziale da mostrare all'utente.

L'oggetto askBattle si occupa di creare la socketUDP e di mettersi in attesa di pacchetti. In più si occupa di creare, quando una richiesta di sfida viene rilevata, un oggetto della classe Concurrent (comunicandogli il riferimento dell'oggetto Interfaces e il nome dello sfidante), prima di rimettersi in attesa della prossima richiesta di sfida.

La classe Concurrent si occupa di chiedere conferma di accettazione della sfida all'utente richiamando il metodo apposito dell'oggetto della classe Interfaces creato all'inizio dal main.

La classe Interfaces invece è la classe che gestisce l'interfaccia grafica, l'interazione con l'utente, gli eventi generati dai suoi comandi.

Contiene 3 metodi per generare interfacce. Il primo genera quella per l'accesso dove si chiedono i dati di registrazione, il secondo genera quella dove l'utente può gestire il suo account, aggiungendo amici, ecc.., il terzo genera l'interfaccia mostrata all'utente durante una battaglia.

Nella stessa classe sono definiti gli ascoltatori degli eventi e la gestione degli eventi stessi. È questa classe che contiene i metodi che implementano il protocollo di comunicazione TCP con il server.

Classi usate dal server

Le classi usate dal server sono: Battle.java, Dictionary.java, ManagerBattles.java, ManagerFileFriends.java, ManagerFileRegistrations.java, ManagerServer.java, ManagerUsers.java, ParametersBattle.java, Registration.java, RemoteRegistration.java, Server.java, Translates.java, User.java, Worker.java, WorkerOnFile.java

La classe Server.java è la classe che contiene il main. Può prendere in input 4 parametri (il path file con gli utenti, il path del file con i punteggi e le relazioni di amicizia, il path del file dizionario e l'encoding) oppure nessuno.

Quando viene eseguita crea un oggetto per ogni classe ManagerUsers, ManagerFileRegistrations, ManagerFileFriends, ParametersBattle, ManagerBattles.

La classe ManagerUsers è la classe per gestire gli utenti attivi o i cui dati subiscono una modifica. La classe ManagerFileRegistrations si occupa di gestire il file delle registrazioni degli utenti. La classe ManagerFileFriends si occupa di gestire il file con le relazioni di amicizia e il punteggio dell'utente. La classe ParametersBattle contiene i parametri K, X, Y, Z, T1, T2 necessari a una sfida. La classe ManagerBattles è quella usata per la gestione delle sfide degli utenti.

La classe ParametersBattle contiene solo metodi statici. Questa classe è stata creata per rendere immediatamente recuperabili i valori dagli oggetti di tutte le altre classi mantenendo la coerenza in modo semplice (non c'è il rischio di modifica di un valore durante un passaggio ai vari oggetti visto che viene direttamente preso dalla classe che ne contiene il valore originale).

Di queste classi viene creata una sola istanza e non ne viene mai creata una seconda.

Dopo aver creato questi oggetti il server fa un controllo di validità sui valori necessari a una sfida (controlla se il numero di parole da tradurre è >0, se il dizionario contiene parole sufficienti per poter poter eseguire una sfida, ecc..)

Successivamente viene creato e pubblicato l'oggetto remoto, viene creata un'istanza della classe che si occupa di fare il backup su file delle relazioni di amicizia e dei punteggi (WorkerOnFile) e un'istanza della classe che si occuperà dei comandi dati al server da parte dell'utente da linea di comando (ManagerServer). Infine viene creata la socket per la connessione TCP e il pool che risponderà alle richieste dei client.

La classe MangerUsers gestisce quindi oggetti della classe User, mentre la classe MangerBattles gestisce oggetti della classe Battle e crea un oggetto della classe Dictionary e uno della classe Translates.

La classe Dictionary si occupa di leggere il dizionario mentre la classe Translates memorizza le parole del dizionario con le relative traduzioni.

L'interfaccia Registration contiene i metodi dell'oggetto remoto messi a disposizione dei client e la classe RemoteRegistration implementa questa interfaccia e quindi questi metodi.

La classe Worker definisce il comportamento del server dopo che riceve messaggi dal client.

Istruzioni e comandi

Tutti i file necessari per eseguire il progetto sono stati forniti nel .zip.

Compilare ed eseguire server e client:

1- Importare il jar di simple json su eclipse
   - Cliccare il tasto destro del mouse sul progetto in cui si vuole importare il punto jar.
   - Cliccare su "Build path".
   - Cliccare su "Configure build path"
   - Selezionare il tab "Libraries"
   - Cliccare "Add JARs" oppure "Add External JARs" se il .jar non si trova in nessun altro progetto
   - Selezionare il .jar da importare
   - Premere "Apri" o "OK" in base alla scelta del .jar interno o esterno
   - Infine premere "Ok"

2- Portare tutti i file .java su Eclipse e salvare.
3- Scegliere il path per i file Utenti.json, Amicizie.json, Dizionario.txt e per la cartella con le immagini forniti.
4- Eseguire la classe "Server" per avviare il server passando 4 argomenti:
   - il path dove trovare o creare il file "Utenti.json" (esempio "./src/progetto/Utenti.json").
   - il path dove trovare o creare il file "Amicizie.json" (esempio "./src/progetto/Amicizie.json").
   - il path dove trovare il file "Dizionario.txt" (esempio "./src/progetto/Dizionario.txt").
   - la codifica da utilizzare (esempio "Windows-1252").

   Questi parametri devono essere passati tutti, non solo alcuni, oppure se non ne viene passato nessuno vengono usati i valori di default che sono quelli negli esempi.

5- Eseguire la classe client per avviare un client passando 2 argomenti:
   - il path dove trovare la cartella con immagini (esempio: "./src/progetto/immagini/");
   - la codifica da utilizzare (esempio "Windows-1252").

Questi parametri devono essere passati tutti, non solo alcuni, oppure se non ne viene passato nessuno vengono usati i valori di default che sono quelli negli esempi.

In questo modo i .java vengono compilati in automatico.

È prevista l'esecuzione di un solo server alla volta, mentre si possono eseguire più client contemporaneamente.

Come eseguire le operazioni del server

Il server accetta 3 operazioni da linea di comando:

- `help: per ottenere la lista di possibili comandi;`
- `backup: per fare un backup dei dati che non vengono memorizzati immediatamente su file;`
- `shutdown: per chiudere il server in modo controllato. Il server non accetta più connessioni da parte di client ma prima di terminare aspetta che tutti gli utenti connessi facciano il logout.`

Per eseguirle scrivere il nome dell'operazione voluta sulla console e premere invio.

Dopo la richiesta di shutdown il server non accetta più nessun altro comando. Questa operazione prevede un backup prima della chiusura.

Come eseguire le operazioni del client

Per poter proseguire in caso di messaggi pop-up, premere "ok", X oppure fare una scelta quando richiesto.

Per il client sono disponibili le seguenti operazioni (tramite interfaccia grafica) fuori da una sfida:

1- Registrazione di un utente
   Eseguire un client, inserire il nickname e la password con cui ci si vuole registrare (entrambe non vuote e non più lunghe di 20 caratteri) e cliccare il pulsante "Registrati".
2- Login
   Eseguire un client. Registrarsi oppure inserire nickname e password con cui ci si era precedentemente registrati e cliccare il pulsante "Login".
3- Logout
   Possibile fuori da una sfida cliccando la X in alto a destra oppure dopo il login cliccando sulla 5 figura nella barra in alto e poi sul pulsante "ESCI".
4- Aggiungi un amico
   Fare il login, cliccare sulla seconda icona nella barra in alto, inserire il nickAmico di colui che si vuole aggiungere, cliccare il pulsante "Aggiungi"
5- Visualizza la lista di amici
   Fare il login, cliccare sulla terza icona nella barra in alto. Se la lista non entra nella schermata è possibile lo scroll.
6- Mostra classifica
   Fare il login, cliccare sulla quarta icona nella barra in alto. Se la classifica non entra nella schermata è possibile lo scroll.
7- Sfida qualcuno
   Fare il login, inserire il nickAmico di colui che si vuole sfidare, cliccare il pulsante "Sfida".

8- Visualizzare il proprio punteggio
   Fare il login. Posizionarsi nel primo tab.

All'interno di una sfida sono possibili le seguenti operazioni (tramite interfaccia grafica):

1- Uscire dalla sfida senza terminarla
   Cliccare sulla X in alto a destra
2- Passare alla parola successiva non rispondendo a quella attuale
   Cliccare il pulsante "Salta"
3- Mandare la traduzione della parola e passare alla successiva
   Cliccare il pulsante "Invia"