Word Quizzle

Relazione progetto finale

TOMMASO MACCHIONI

Università di Pisa

Indice

1 Introduzione

Lo scopo di questo progetto è stato quello di sviluppare un'applicazione, in linguaggio Java, al fine di utilizzare le conoscenze acquisite durante il corso di "Reti di Calcolatori e Laboratorio" presso l'Università di Pisa.

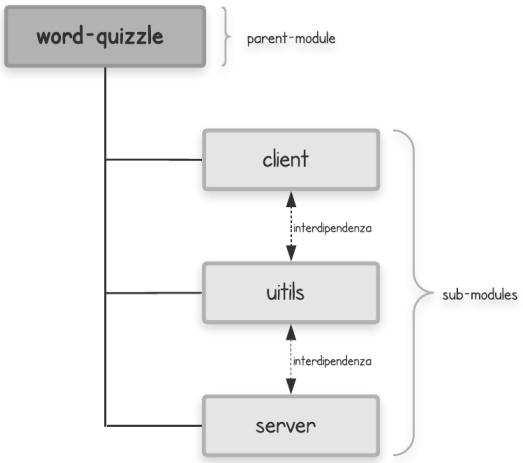
Il progetto consiste nell'implementazione di un sistema di sfide di traduzione italiano-inglese tra utenti registrati al servizio. Gli utenti registrati possono sfidare i propri amici ad una gara il cui scopo è quello di tradurre in inglese il maggiore numero di parole italiane proposte dal servizio. Il sistema consente inoltre la gestione di una rete sociale tra gli utenti iscritti. L'applicazione è implementata secondo una architettura client-server.

Per descrivere al meglio le specifiche ed il funzionamento dell'intera applicazione è stato scelto di adottare una strategia mista in cui verrà dapprima definito uno schema scheletrico dell'architettura generale, successivamente questo verrà decomposto in sottoschemi e dettagliati singolarmente, ed infine, facendo uso di use cases specifici, quest'ultimi verranno integrati in modo da comprendere appieno la loro interazione ed avere dunque una visione d'insieme più semplice e chiara possibile.

2 Architettura generale

L'intera applicazione è stata realizzata utilizzando *Java Development Kit* versione 13 (*JDK13*) ed è stato utilizzato *Maven*: un framework per la gestione di progetti software basati su Java, il quale, seguendo il paradigma "Convention over Configuration", ha permesso di semplificare e rendere più rapida la sua realizzazione.

Tra le caratteristiche più utili che il framework fornisce abbiamo la possibilità di organizzare il codice in *moduli* distinti ma interdipendenti, e di importare le librerie esterne grazie all'affidamento di repository disponibili in rete.



Nello specifico, l'architettura generale consiste in un progetto Maven multi-modulo in cui tre moduli (`utils`, `server`, `client`), detti *sub-module*, fanno capo a un modulo genitore (`word-quizzle`), detto *parent-module*.

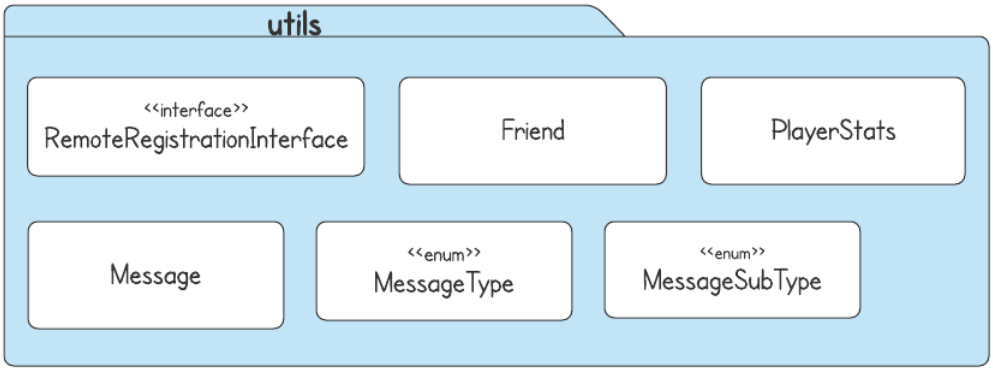
Il modulo `word-quizzle` ospiterà i suoi sotto moduli e solo quest'ultimi conterranno il codice sorgente dell'intera applicazione e le risorse da essa utilizzate.

Nei prossimi paragrafi verranno evidenziate le scelte implementative e quindi il design, adottati nei singoli moduli e, per una maggiore chiarezza espositiva, l'ultimo paragrafo racchiuderà gli screenshot della GUI sviluppata.

3 Modulo "utils"

Questo modulo, come suggerisce il nome, contiene quelle classi comuni e di utilità, sia al modulo client che a quello server, così da sfruttare al meglio la pratica del riuso del codice.

Classi implementate:



- `RemoteRegistrationInterface`. L'interfaccia che il server implementerà per poi esportare l'oggetto remoto e registrarlo nel registry;
- `Friend`. Classe per rappresentare un amico. Contenente il proprio nickname, score e stato: *online*, *offline* o *playing*;
- `PlayerStats`. Struttura dati utilizzata per rappresentare le statistiche di un singolo giocatore durante una partita.
- `Message`. Classe utilizzata nel protocollo client-server.
- `MessageType` e `MessageSubType`. Classi di tipo *enumarate* utilizzate nell'identificazione del tipo di messaggio scambiatosi tra server e client.

L'utilizzo delle singole classi verrà approfondito nei paragrafi successivi quando verrà trattata l'interazione client-server ed il protocollo adottato.

4 Modulo "client"

4.1 Introduzione

Per implementare l'applicazione che l'utente finale (il giocatore) utilizzerà, è stato fatto uso di *JavaFX* versione 13: una libreria grafica che consente di sviluppare *Rich Client Application*, ovvero delle applicazioni dotate di un'interfaccia utente costruita attraverso un insieme di componenti predefiniti. JavaFX ha l'intento di rimpiazzare Swing come libreria GUI standard per Java.

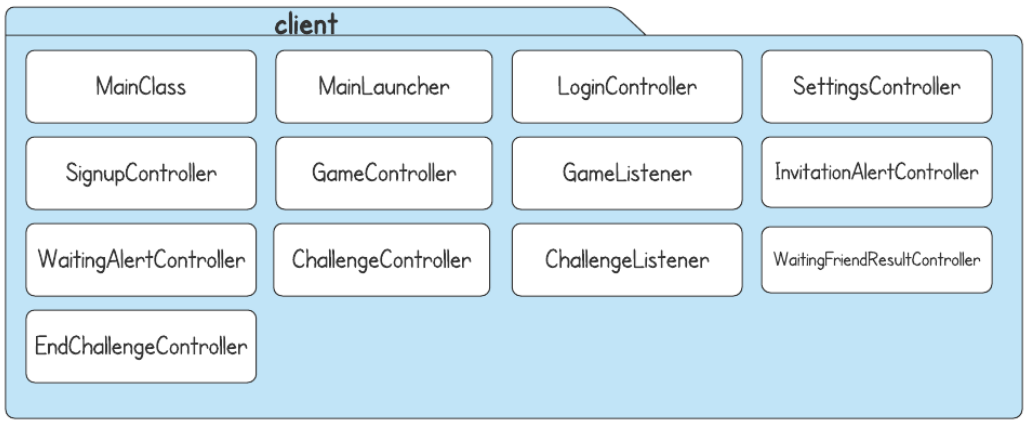
Grazie all'utilizzo di *FXML*: un linguaggio di markup, basato su XML, che consente la definizione delle interfacce grafiche in modo dichiarativo, è stata mantenuta una netta separazione del layer di presentazione dalla logica applicativa.

Per la realizzazione di ogni singolo file FXML, chiamato *view*, è stato utilizzato *Scene Builder*: un *visual layout tool* per la realizzazione di interfacce utente, e per la modifica del *look and feel* di ogni view si è fatto uso di fogli di stile CSS.

Ogni *view* rappresenta una *scena* e ad ogni scena è associata una classe Java, chiamata *controller*, utilizzata per la gestione degli eventi collegati ad essa. Gli eventi registrati in una view sono in corrispondenza biunivoca con i metodi implementati nel suo controller corrispondente.

A meno di eccezioni l'utente visualizzerà una scena per volta.

Classi implementate:



- `MainClass`. Ha il mero compito di chiamare il comando `main` della classe `MainLauncher`.
- `MainLauncher`. Inizializza e visualizza la prima scena di login.
- `*Controller`. Qualsiasi classe che termina con "Controller" rappresenta la classe associata ad una specifica view e quindi scena.
- `GameListener` e `ChallengeListener`. Sono classi che implementano l'interfaccia `Runnable`. Sono i thread che il client utilizzarà e rappresentano dunque il cuore dell'applicazione e nei quali avviene la comunicazione con il server e con i vari controllers.

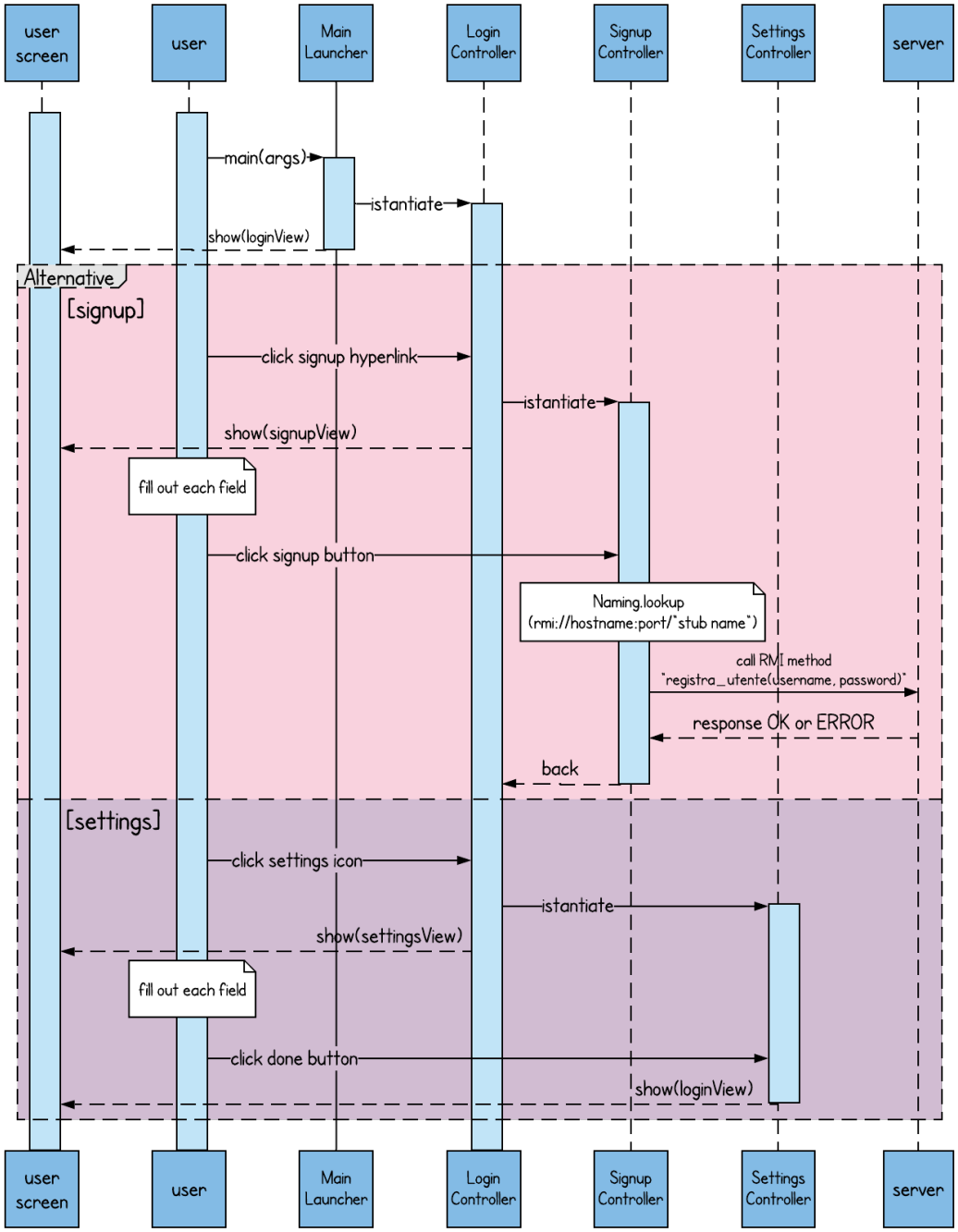
La comunicazione tra i thread ed i controllers, in realtà, si basa semplicemente sul chiamare uno i metodi dell'altro. In sostanza, quando l'utente interagirà con una scena, verrà scatenato un evento e quindi un metodo della classe controller ad essa associato e quest'ultimo chiamerà un metodo statico del thread. Viceversa, quando un thread riceve una messaggio dal server, potrebbe essere necessario aggiornare la GUI e quindi chiamare un metodo del controller della scena attualmente attiva. Un ulteriore compito dei thread e dei controller sarà quello di passare da una scena all'altra.

4.2 Scenari

Per comprendere meglio la relazione che intercorre tra view, controller e i thread istanziati, e per discutere i diversi possibili scenari dell'applicazione client, ci avvarremo di diagrammi di sequenza UML. Da notare che una chiamata al metodo *show("nome file FXML")* implica la visualizzazione di una determinata scena che è stata omessa per motivi di impaginazione ma che è consultabile nell'ultimo paragrafo nella sezione dedicata alla GUI.

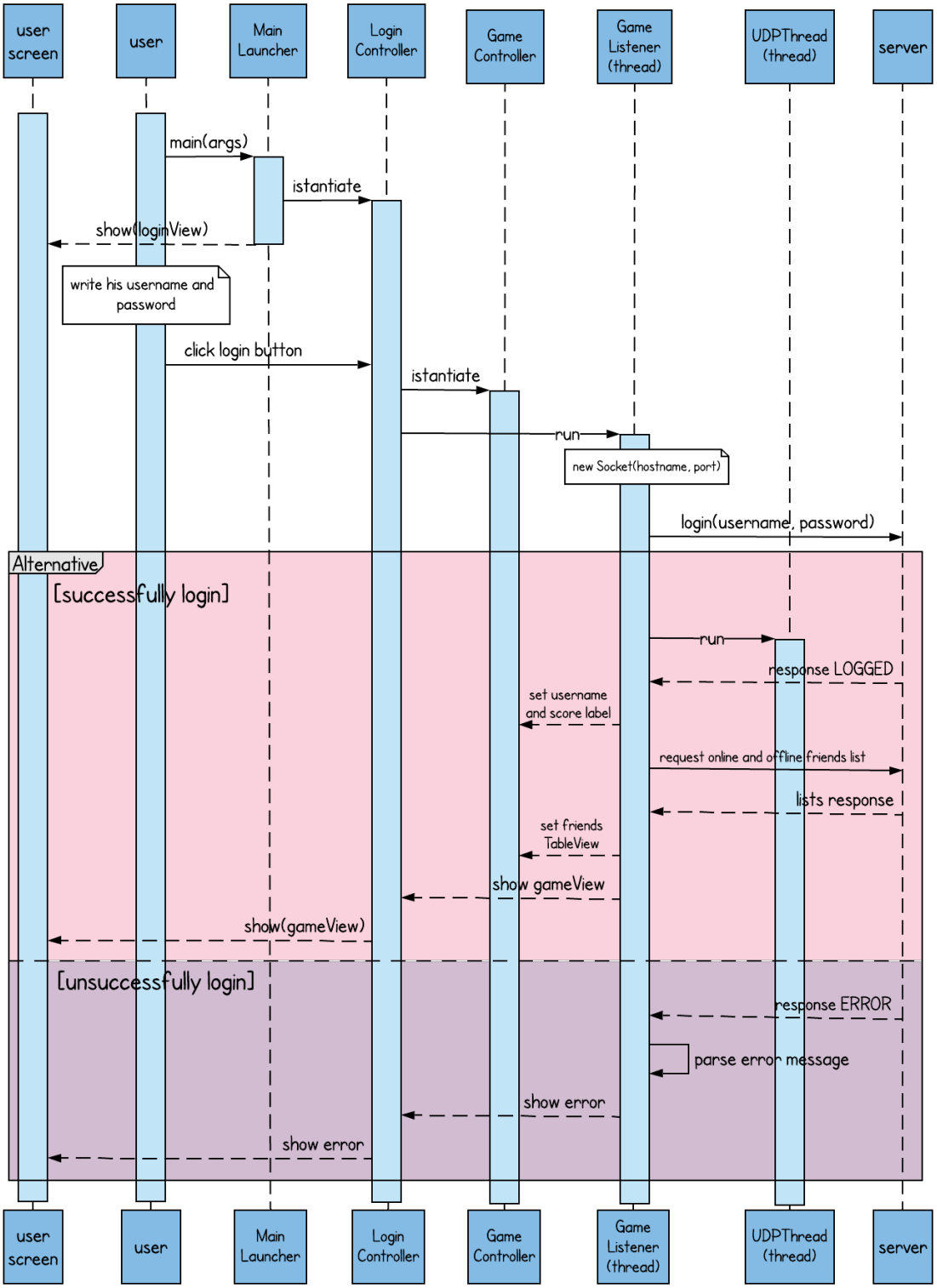
4.2.1 Signup e settaggio indirizzo server

Vediamo il caso in cui l'utente, dopo aver avviato l'applicazione, esegue la procedura di registrazione o alternativamente modifica le impostazioni di indirizzo e porta del server e del registro RMI:



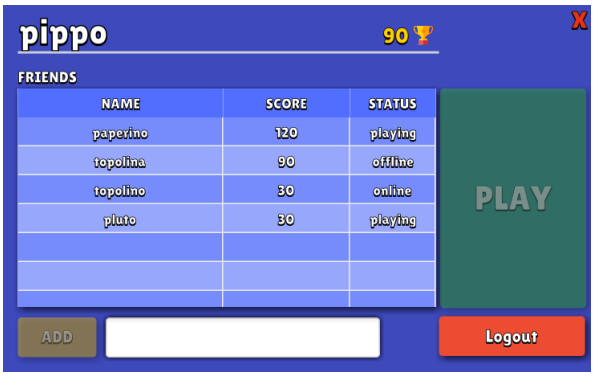
4.2.2 Login

Una terza alternativa rappresenta la procedura di login:



`GameListener` sarà dunque il thread incaricato di ricevere e inviare i messaggi da e verso il server durante la visualizzazione della scena *gameView* mentre `UDPThread` rappresenta un thread privato istanziato da `GameListener` e, come vedremo in seguito, servirà per la ricezione di messaggi di sfida da parte di altri giocatori.

Un utente, una volta effettuato correttamente l'accesso, visualizzerà, ad esempio, la seguente scena:



Essa rappresenta la finestra visualizzata da un utente di nome *pippo* con un punteggio accumulato di 90 punti, che ha come amici: *paperino*, *topolina*, *topolino* e *pluto*, ognuno con il proprio punteggio accumulato e lo stato attuale: *online*, *offline* o *playing*.

Per una migliore esperienza per l'utente finale, è stato scelto di visualizzare in maniera permanente l'elenco dei propri amici ed aggiornare la lista qualora uno di loro si collegasse, scollegasse o stesse giocando.

4.2.3 Notifiche di stato

L'aggiornamento di stato di un proprio amico avviene grazie al thread `GameListener` che, in ascolto del server, riceverà eventuali notifiche di cambio stato dei propri amici. Lo stesso vale quando il client effettua il login, il logout oppure comincia una nuova partita: verrà inviato un messaggio di notifica di aggiornamento di stato al server e quest'ultimo comunicherà ai suoi amici l'aggiornamento.

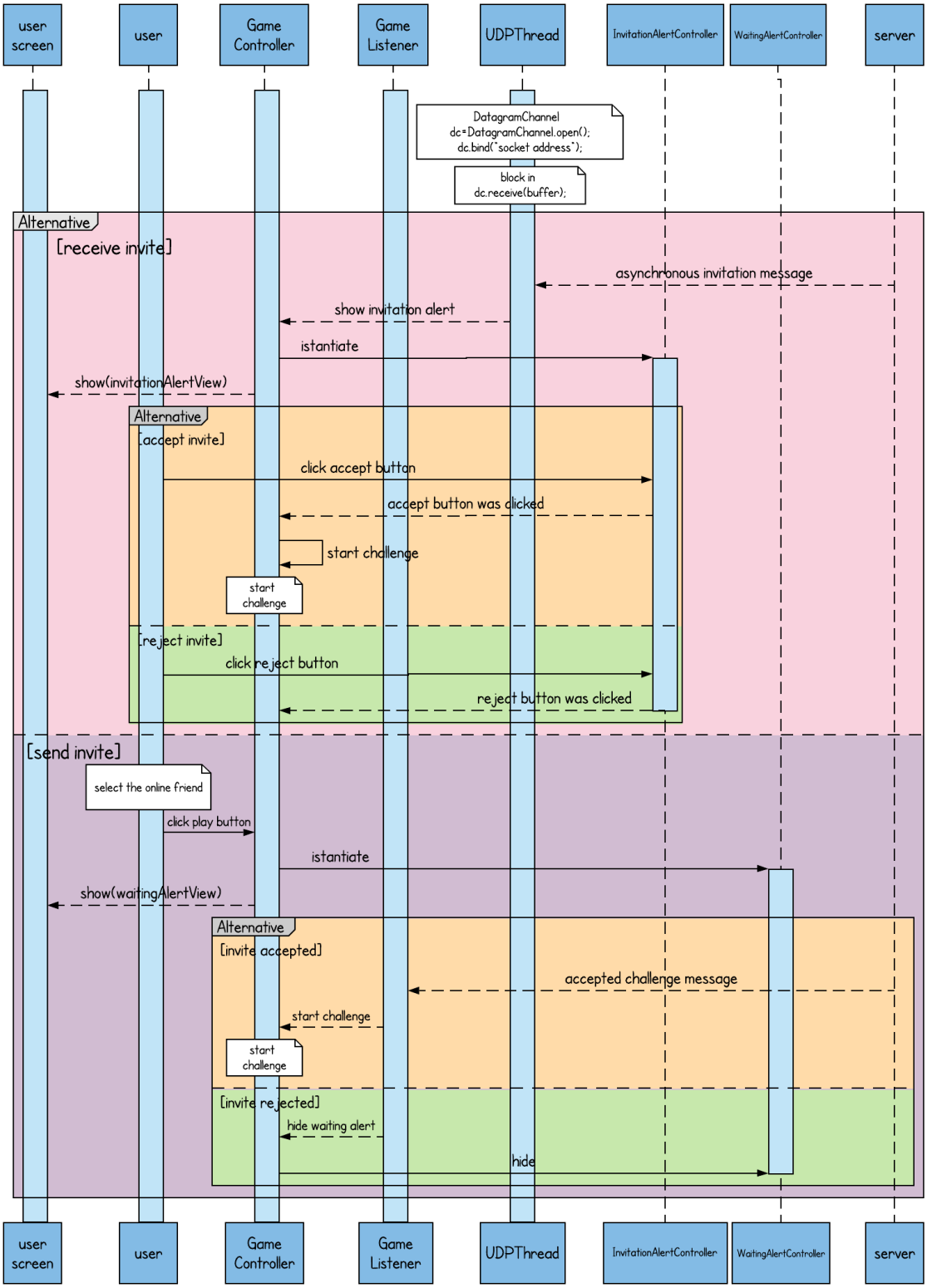
4.2.4 Aggiunta di un amico

Per aggiungere un nuovo amico basterà scrivere il suo nome nel campo di testo e cliccare sul buttone *ADD*. Questo scatenerà un evento e quindi un metodo della classe `GameController` che a sua volta chiamerà un metodo statico del thread `GameListener` che invierà il messaggio al server. A questo punto, `GameListener`, in attesa di risposta, riceverà un messaggio positivo oppure negativo:

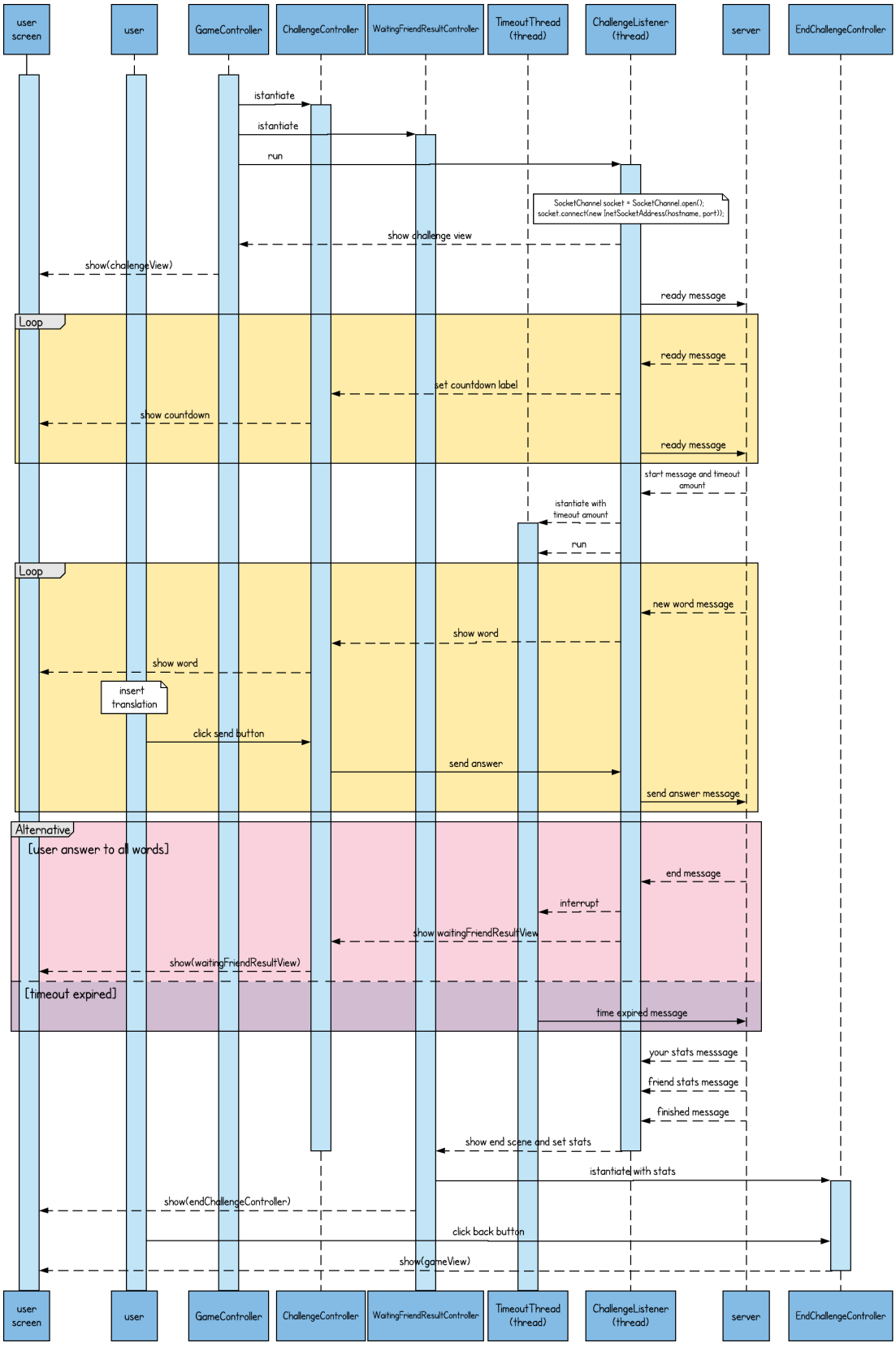
- **Caso positivo**. Verrà aggiunta una riga alla Table View contenente le informazioni del nuovo amico: nome, punteggio e stato attuale.
- **Caso negativo**. Verrà visualizzato un messaggio di errore. Gli errori riscontrabili sono causati da: utente non presente nel database oppure utente già nella lista di amici.

4.2.5 Sfida

Prima di illustrare il meccanismo dietro al sistema di sfida è necessario comprendere come avviene l'invio e la ricezione di un invito di sfida. L'accettazione di questa, sarà condizione necessaria e sufficiente per avviare la sfida. (Nota: si suppone che l'utente sia nella situazione in cui abbia effettuato l'accesso e stia visualizzando una finestra come quella della figura precedente, ovvero la *gameView*).



Vediamo adesso la sequenza di operazioni svolte dall'applicazione client una volta che la sfida è stata accettata sia dopo la ricezione di un invito che per una richiesta di sfida da lui effettuato. In altre parole, il client esegue le medesime operazioni in entrambi i casi.



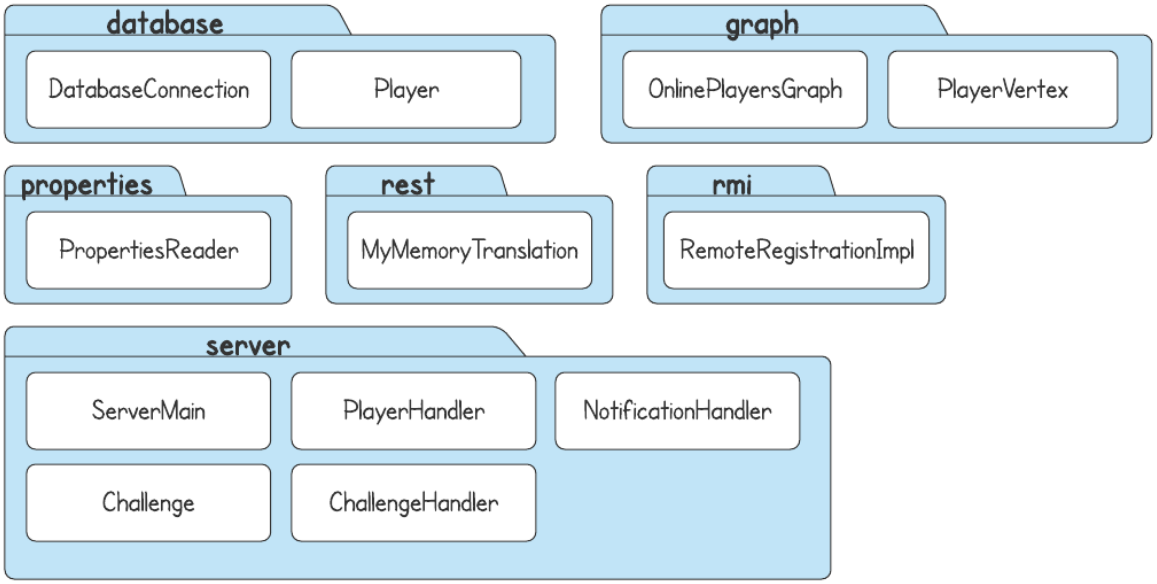
5 Modulo "server"

5.1 Introduzione

Per implementare l'applicazione server è stato fatto uso di librerie esterne quali:

- *Gson.* Per convertire Java Object nella loro rappresentazione JSON e viceversa, ed è stata utilizzata per implementare la persistenza delle informazioni di registrazione, relazioni di amicizia e punteggio degli utenti su file json;
- *JGraphT.* Per implementare una struttura dati a grafo utilizzata nella rappresentazione degli utenti attualmente online.

Il modulo server è stato suddiviso in 6 packages differenti:



Le risorse impiegate sono tre:

- `config.properties` Per memorizzare proprietà del server quali: indirizzo del server, porta, porta RMI, lunghezza massima per username e password e ulteriori proprietà per quanto concerne la sfida.
- `database.json` Per memorizzare gli utenti registrati.
- `dictionary.txt` Per memorizzare la lista di parole italiane che il server utilizzerà in fase di sfida.

Nel paragrafo che segue verranno esaminati i packages singolarmente.

5.2 Packages

5.2.1 database

Il package *database* è composto da due classi:

- `DatabaseConnection`. Questa classe viene utilizzata per la memorizzazione o il fetching dei dati nel file json `database.json`. Implementa due tipologie di design patter:
  - *Utility Class Pattern.* Perché non ha un proprio stato, tutti i metodi sono statici e fornisce metodi per altre classi.
  - *Java Singleton Pattern con lazy initialization.* Perché esisterà una sola istanza di questa classe nella JVM e l'istanzazione avverrà alla prima chiamata di uno dei suoi metodi.

  Per la scrittura e lettura del file json sono stati utilizzati due file stream: `BufferedWriter` e `BufferedRead`, oltre alla libreria esterna *Gson*. Per la serializzazione e deserializzazione sono state utilizzate le classi `Player` e `Friend`. Infine, per la gestione della concorrenza, e quindi evitare stati di inconsistenza, si è fatto uso di lock a livello di classe tramite la keyword `synchronize`.
- `Player`. Questa struttura dati, oltre ad essere utilizzata per la serializzazione/deserializzazione di oggetti json, viene sfruttata dal thread che gestirà il singolo client.

5.2.2 properties

Contiene la sola classe `PropertiesReader`. Anche questa classe implementa i due design pattern descritti in precedenza per la classe `DatabaseConnection` ed è impiegata nella lettura delle proprietà memorizzate nel file `config.properties`.

5.2.3 rmi

Contiene la sola classe `RemoteRegistrationImpl` ed è l'implementazione dell'interfaccia `RemoteRegistrationInterface` memorizzata nel modulo `utils`. Sarà di fatto l'oggetto remoto che un client andrà ad utilizzare per la fase di registrazione.

5.2.4 rest

Anche questo package contiene una singola classe: `MyMemoryTranslation`. Viene impiegata dal server durante la fase di sfida. Infatti, sfruttando le API fornitaci da `mymemory.translated.net`, l'utilizzo della classe `java.net.URL` e la libreria esterna *Gson*, il server effettuerà richieste HTTP al sito web per ricevere, in formato json, una lista di possibili traduzioni di una data parola.

Inoltre, per una maggiore performance ed efficienza, è stato adottato un meccanismo di *caching* in cui si vengono a memorizzare le traduzioni già effettuate all'interno di una HashMap.

5.2.5 graph

Per rappresentare gli utenti attualmente online si è scelto di utilizzare una struttura dati a grafo non orientato in cui ogni nodo è un'istanza della classe `PlayerVertex` ed il grafo è rappresentato dalla classe `OnlinePlayersGraph`. L'implementazione è stata possibile grazie all'utilizzo della libreria esterna `JGraphT`.

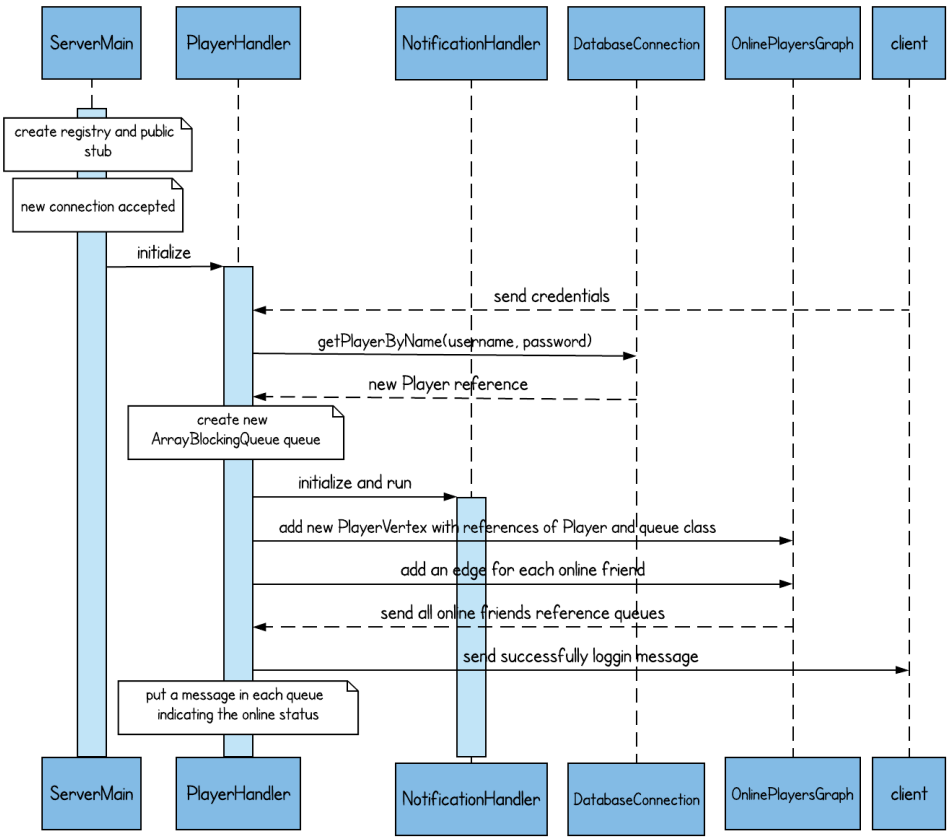
In particolare, `OnlinePlayersGraph` estende la struttura dati, fornitaci dalla libreria, `AsSynchronusGraph` che, tra le caratteristiche più importanti, incapsula meccanismi per la gestione della concorrenza. Inoltre, affinché vi sia soltanto una sola istanza del grafo a runtime, si è adottato nuovamente il *Java Singleton Pattern*.

La memorizzazione degli utenti attualmente online è stata fondamentale per poter applicare i meccanismi di sfida tra amici, inviare ad un giocatore la lista dei propri amici online ed inviare richieste di amicizia. Di fatto, come già accennato precedentemente, ogni nodo è rappresentato dalla classe `PlayerVertex` la quale a sua volta, tra le variabili di istanza, detiene un riferimento al giocatore rappresentato ed un riferimento ad una `ArrayBlockingQueue` per memorizzare messaggi al quale destinare.

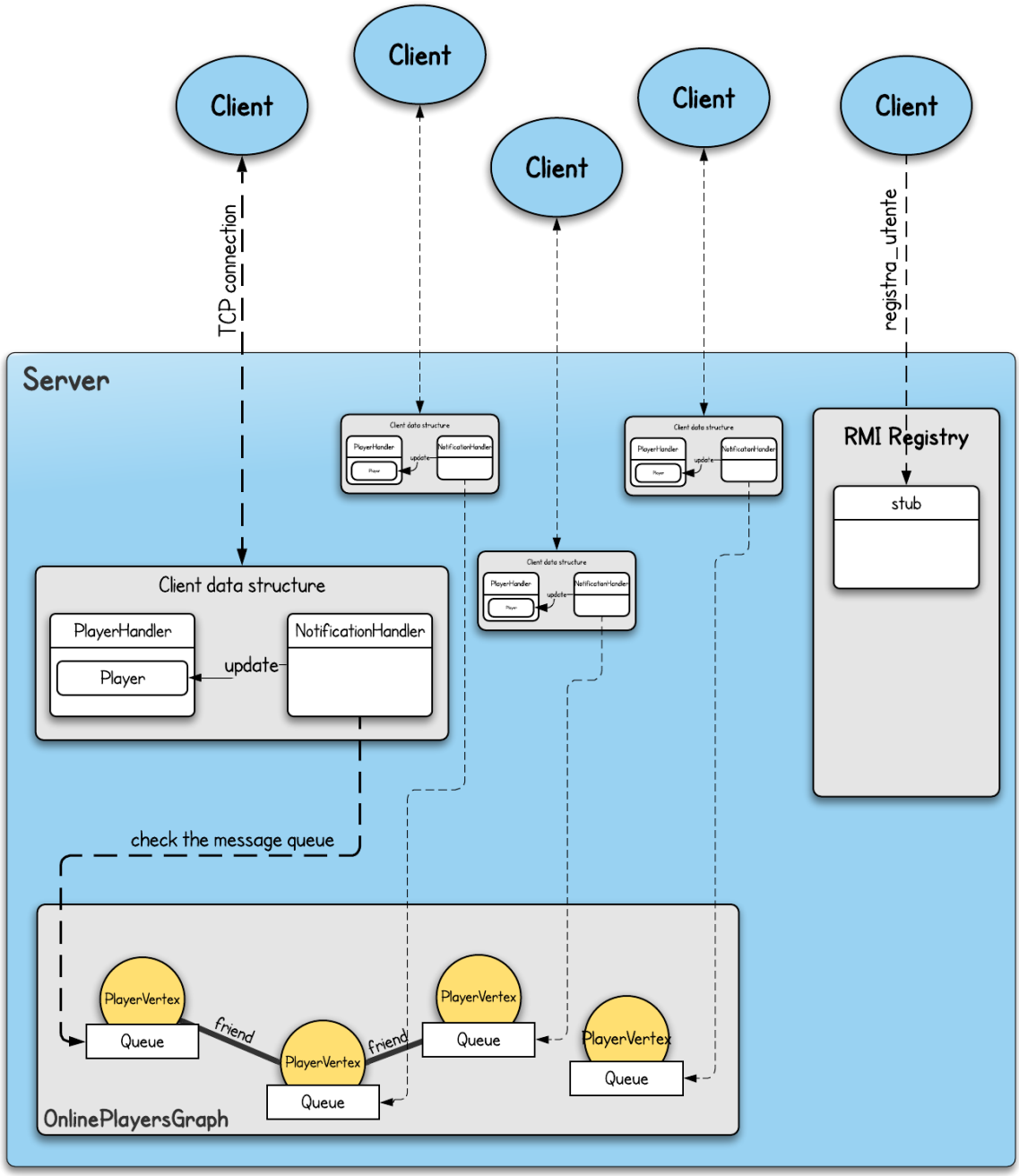
5.2.6 server

La struttura del server è di tipo *multithreaded* in cui ad ogni client ad esso collegato verrà assegnato il thread `PlayerHandler`. Il metodo main è contenuto nella classe `ServerMain` in cui inizialmente viene creato il *registry* e pubblicato lo *stub* dopodiché rimane in ciclo infinito accettando connessioni da parte di client e inizializzando e lanciando il thread a loro assegnato.

Il thread `PlayerHandler` gestisce dunque la comunicazione client-server e una volta effettuato il login con successo, svolge alcune importanti operazioni iniziali esposte nel seguente diagramma:

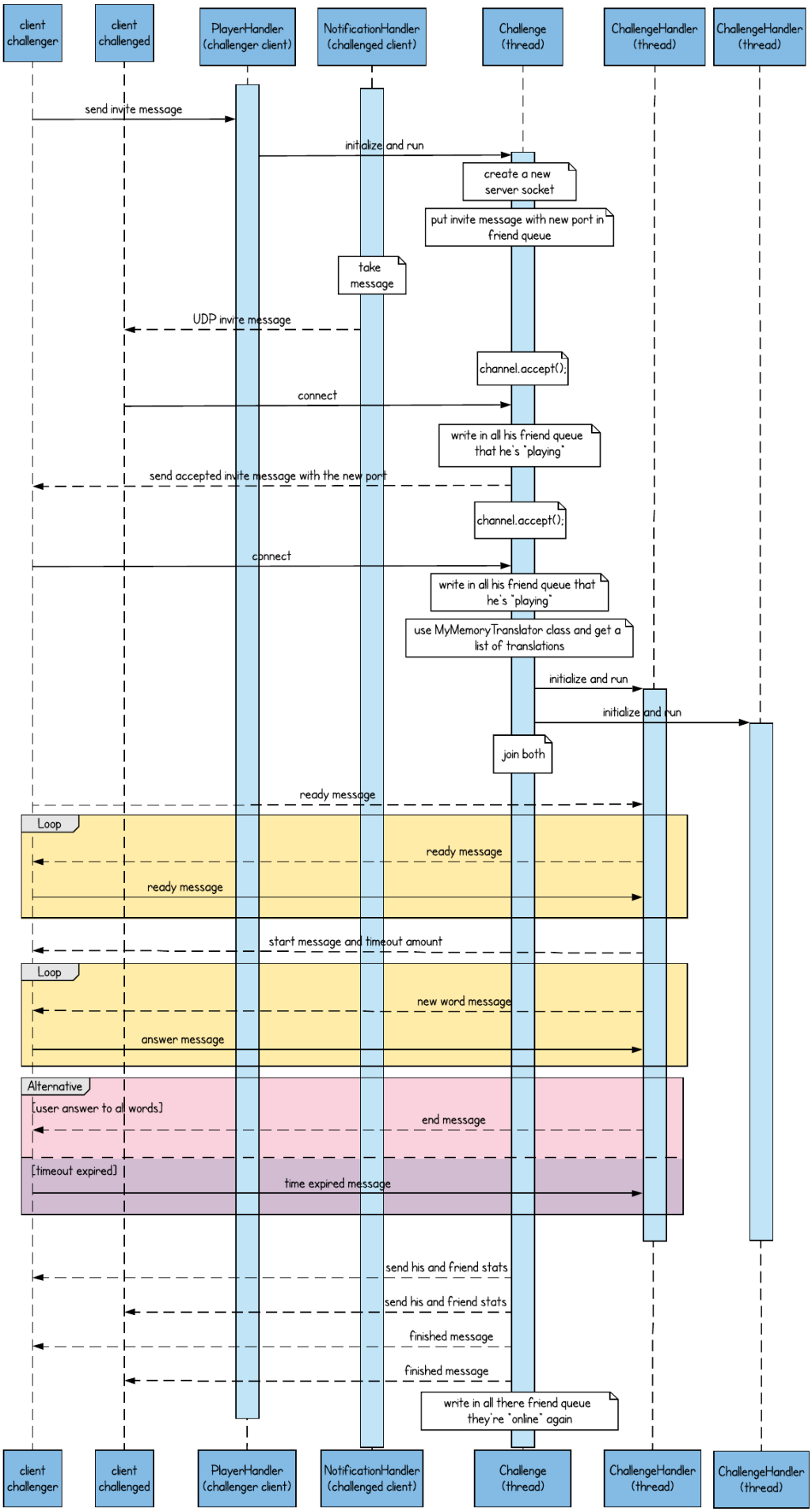


Una visione astratta dell'intera applicazione può essere rappresentata così:



Le altre cassi che il package include sono `Challenge` e `ChallengeHandler`. Sono entrambe adibite alla gestione della sfida tra i due amici. Vediamo ora quali sono i passaggi che intercorrono tra l'invio di sfida di un client e la fine della sfida e per farlo ci avvarremo sempre di diagrammi di sequenza UML.

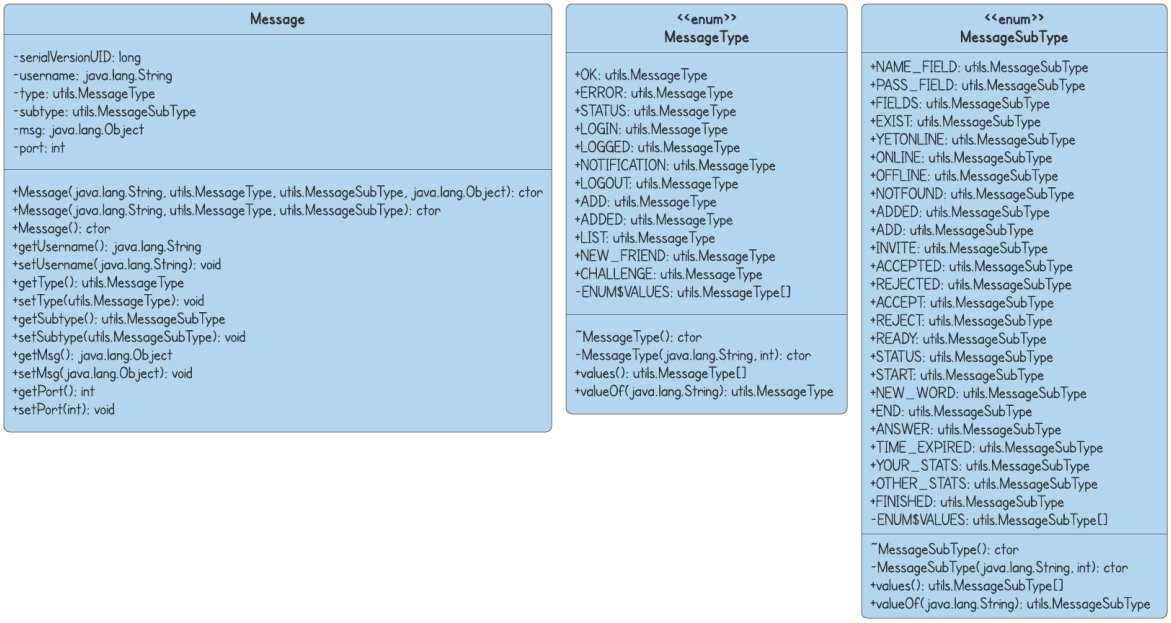
(Nota: per semplicità è mostrata solo la comunicazione tra il client ed il thread `ChallengeHandler`. Nella realtà le stesse operazioni sono svolte in "parallelo" tra il secondo client ed il secondo handler).


client challenger
client challenged
PlayerHandler (challenger client)
NotificationHandler (challenged client)
Challenge (thread)
ChallengeHandler (thread)
ChallengeHandler (thread)
send invite message
initialize and run
create a new server socket
put invite message with new port in friend queue
take message
UDP invite message
channel.accept();
connect
write in all his friend queue that he's "playing"
send accepted invite message with the new port
channel.accept();
connect
write in all his friend queue that he's "playing"
use MyMemoryTranslator class and get a list of translations
initialize and run
initialize and run
join both
ready message
Loop
ready message
ready message
start message and timeout amount
Loop
new word message
answer message
Alternative
[user answer to all words]
end message
[timeout expired]
time expired message
send his and friend stats
send his and friend stats
finished message
finished message
write in all there friend queue they're "online" again
client challenger
client challenged
PlayerHandler (challenger client)
NotificationHandler (challenged client)
Challenge (thread)
ChallengeHandler (thread)
ChallengeHandler (thread)

6 Protocollo

Per poter comprendere al meglio l'interazione tra i clients e il server è necessario introdurre il protocollo impiegato.

Come spiegato nel paragrafo dedicato al modulo "utils", abbiamo tre classi utilizzate per lo scambio di informazioni tra il client ed il server:



La comunicazione tra i due host avverrà tramite connessioni TCP (ad eccezione dell'invio della sfida e della registrazione di un utente) e tramite l'utilizzo di streams. In particolare un oggetto di tipo `Message` verrà serializzato/deserializzato e spedito/ricevuto tramite `ObjectOutputStream`/`ObjectInputStream`.

Le classi `MessageType` e `MessageSubType`, come si può vedere dai diagrammi UML, sono utilizzate per differenziare i vari tipi di messaggio che il client e il server ricevono.

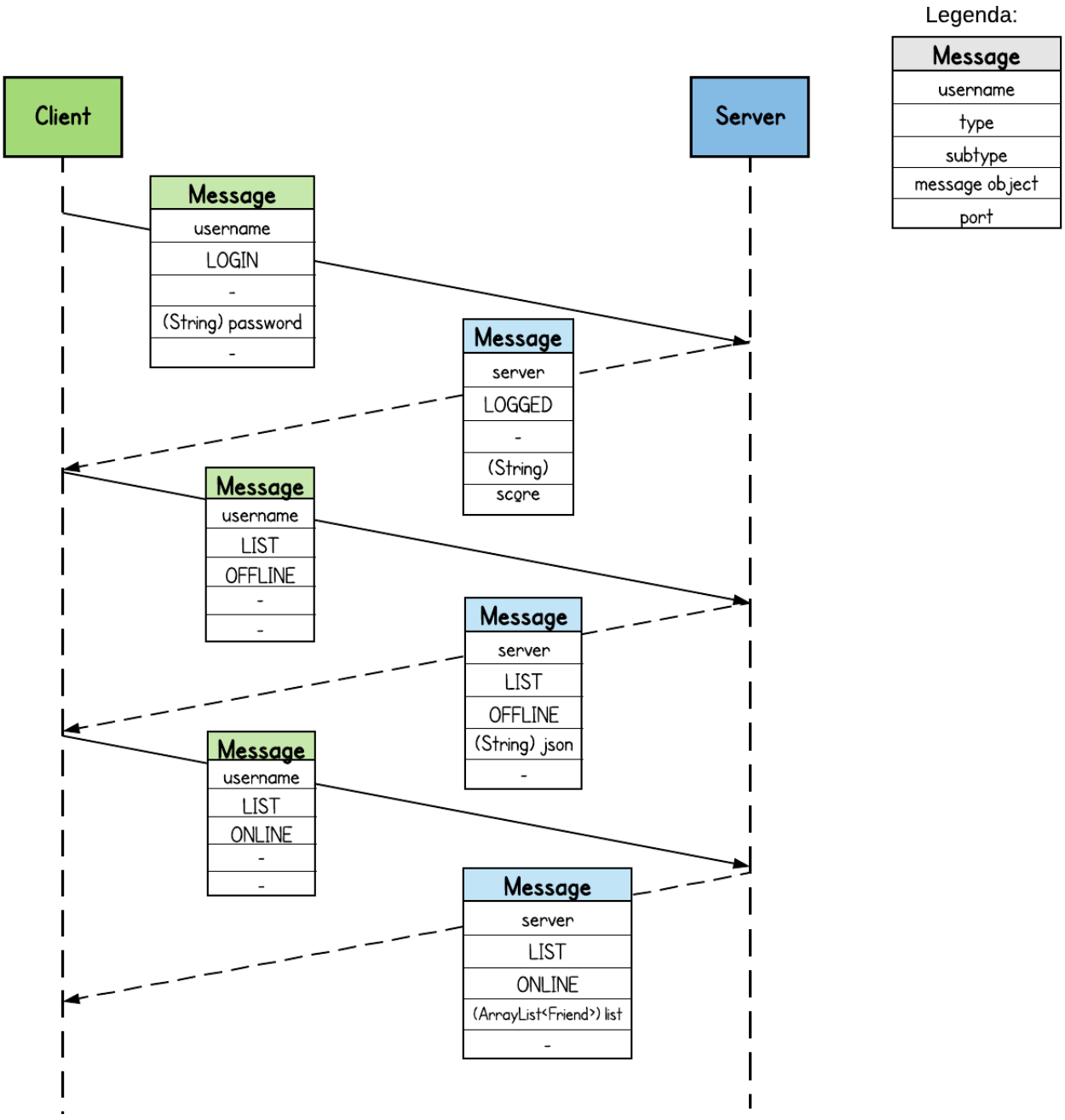
Vediamo adesso le principali variabili della classe `Message`:

- `serialVersionUID`: numero utilizzato per la deserializzazione;
- `username`: il mittente del messaggio;
- `type`: di tipo `MessageType`, identifica il tipo del messaggio;
- `subtype`: di tipo `MessageSubType`, identifica il sotto-tipo del messaggio;
- `msg`: di tipo `Object`, contiene il payload del messaggio;
- `port`: campo utilizzato nel trattamento della sfida.

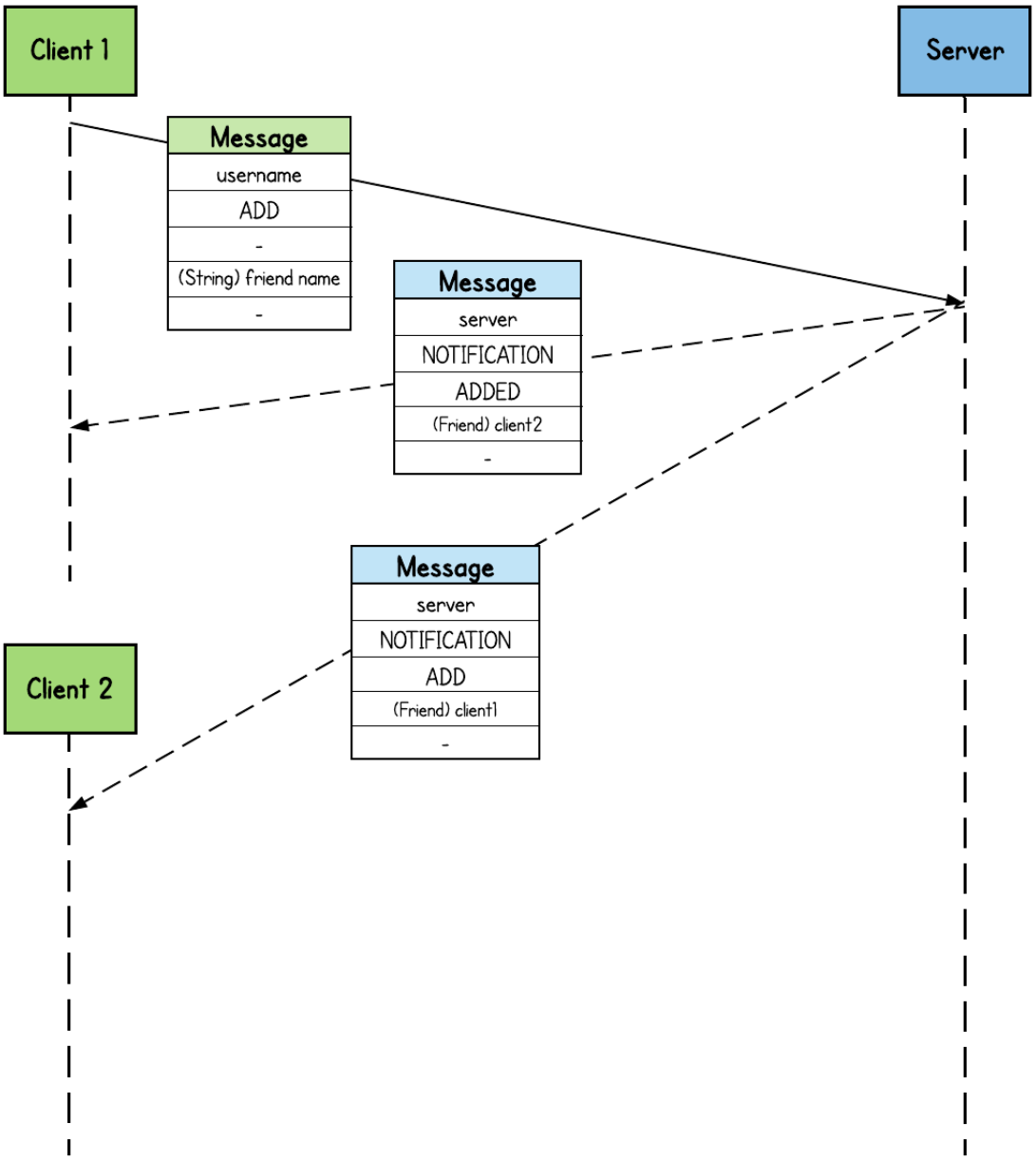
Vediamo adesso tutti gli scenari proposti nei paragrafi precedenti solo dal punto di vista dello scambio dei messaggi.

6.1 Scenari

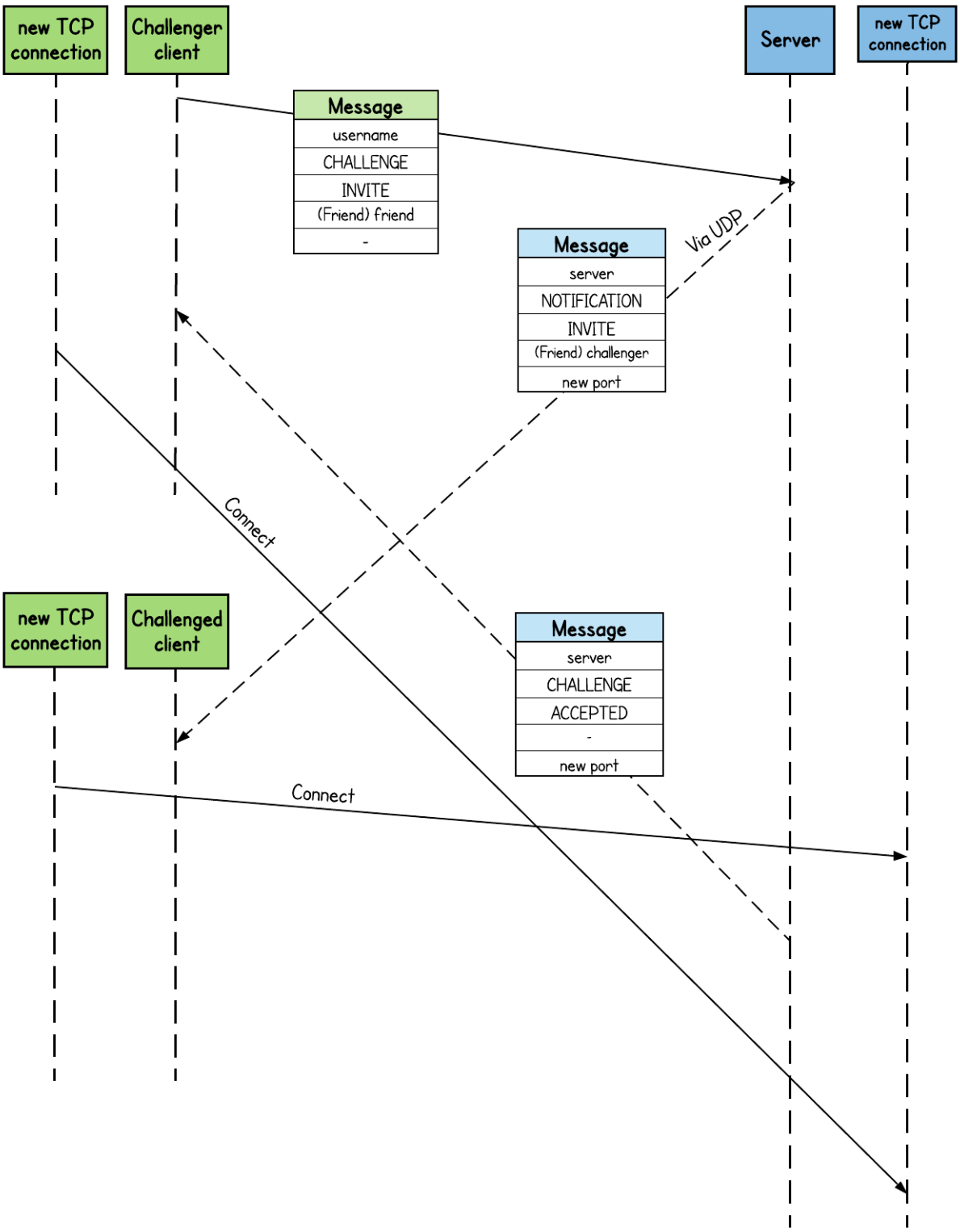
6.1.1 Login



6.1.2 Richiesta di amicizia

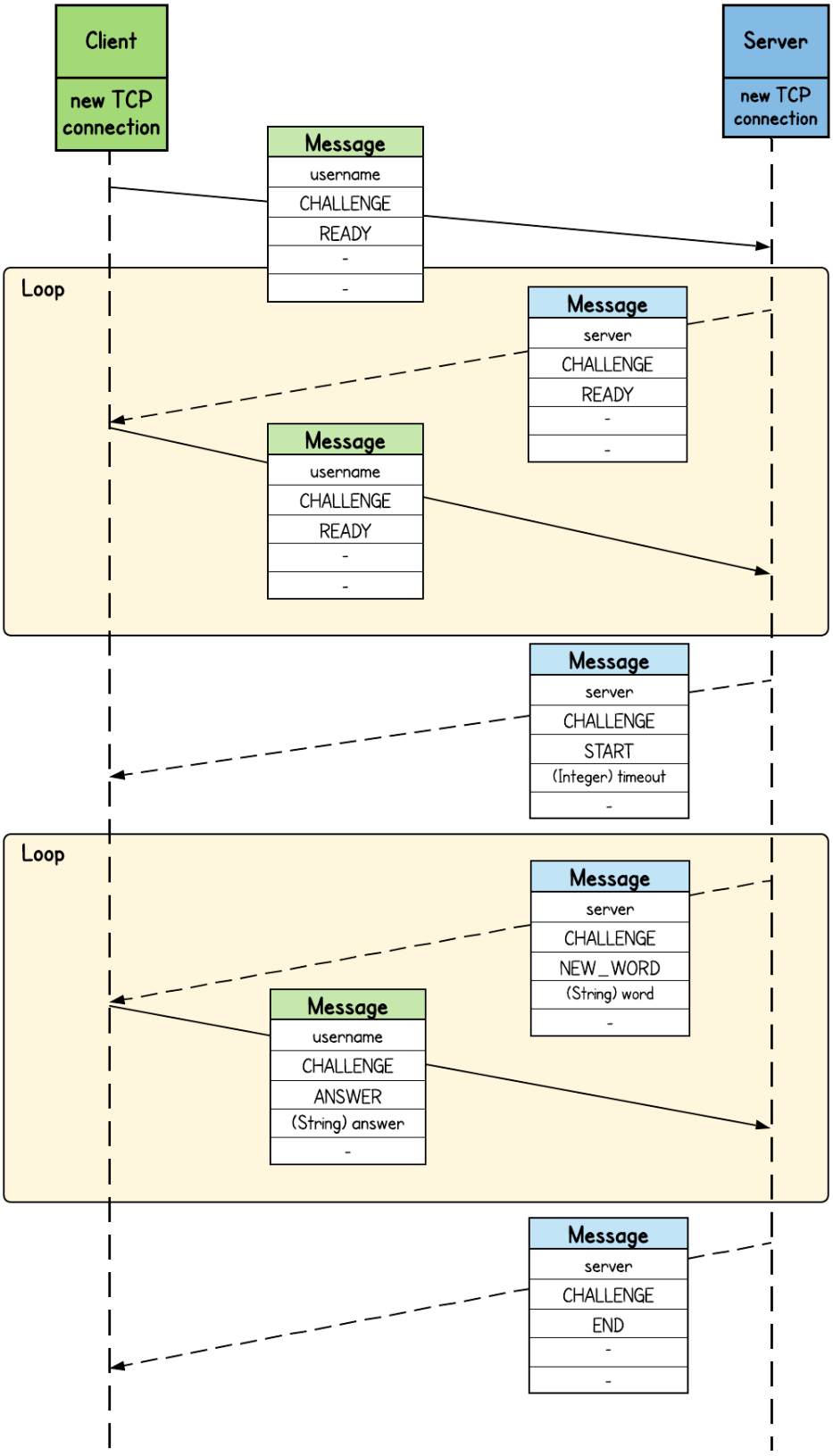


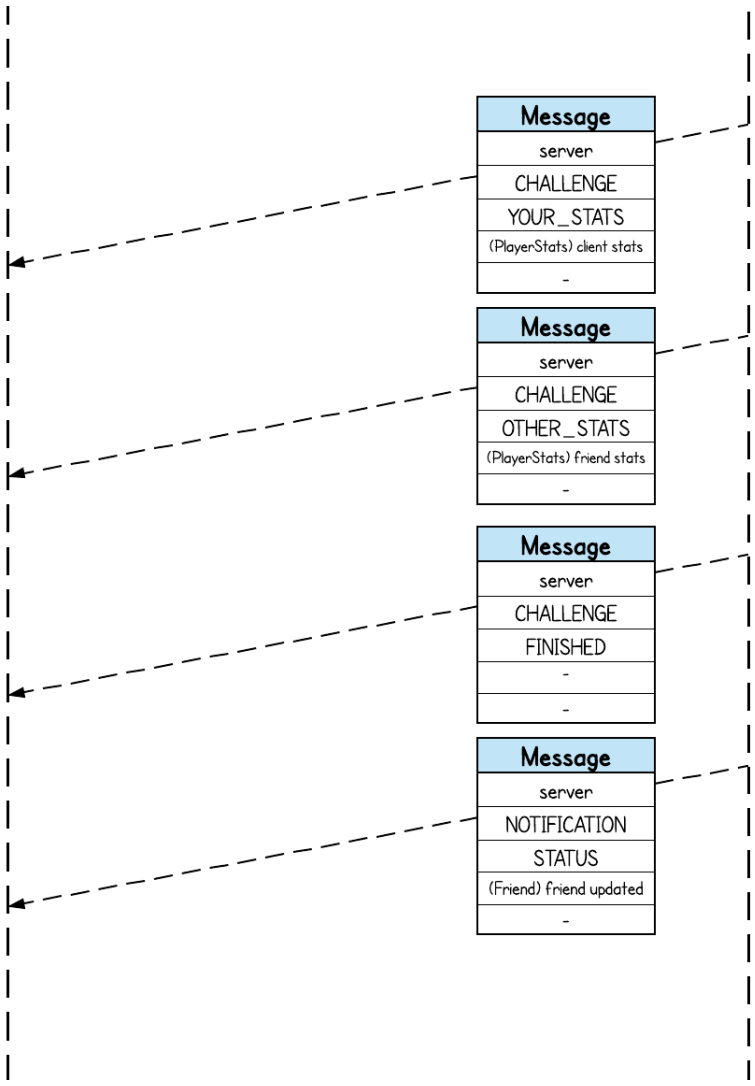
6.1.3 Invito di sfida



6.1.4 Sfida

Al solito mostreremo le comunicazioni solo di un client in quanto l'altro client riceverà e invierà i medesimi messaggi.




Message
server
CHALLENGE
YOUR_STATS
(PlayerStats) client stats
-
Message
server
CHALLENGE
OTHER_STATS
(PlayerStats) friend stats
-
Message
server
CHALLENGE
FINISHED
-
-
Message
server
NOTIFICATION
STATUS
(Friend) friend updated
-

7 Conclusione

Tra le caratteristiche non specificate nella consegna ma che hanno apportato ad ottimi benefici troviamo l'utilizzo della libreria esterna `JGraphT` che ha permesso di gestire agevolmente la concorrenza a livello server ed il meccanismo di caching per le traduzioni che ha permesso di evitare inutili richieste HTTP per traduzioni già effettuate in precedenza. Mentre tra i punti di debolezza dell'intera applicazione, vi è una quasi totale assenza di meccanismi di sicurezza: le password sono inviate, ricevute e memorizzate in chiaro.

In conclusione, questo progetto non ha certamente la pretesa di potersi confrontare con le applicazioni attualmente sul mercato soprattutto sul piano grafico, di performance e sicurezza ma è stato comunque un ottimo esercizio per comprendere cosa risiede alla loro base e la loro complessità.

8 Requisiti e istruzioni

Requisiti:

- JavaSE-13
- Eclipse

Istruzioni:

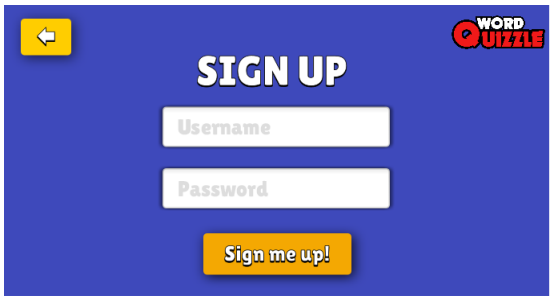
1. Estrarre l'archivio `MACCHIONI_TOMMASO_536402_word-quizzle.zip`
2. In Eclipse: `File->Open Projects from File Systems...`
3. Importare la cartella precedente.
4. Dovrebbero essere stati importati i 4 moduli: il parent e i sotto-moduli. Avviare la fase di `install` di Maven sul modulo parent `word-quizzle`
5. Avviare la classe `MainServer` nel modulo `server`
6. Avviare la classe `MainClass` nel modulo `client`

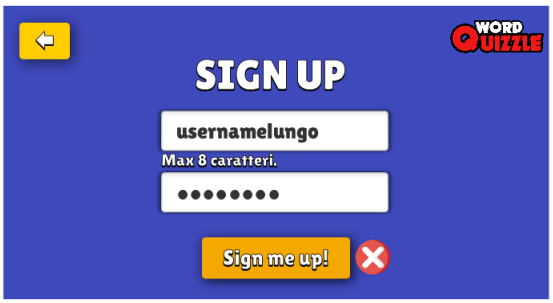
Qualora si volessero modificare le impostazioni del server utilizzare il file `properties.config`.

9 GUI screenshots

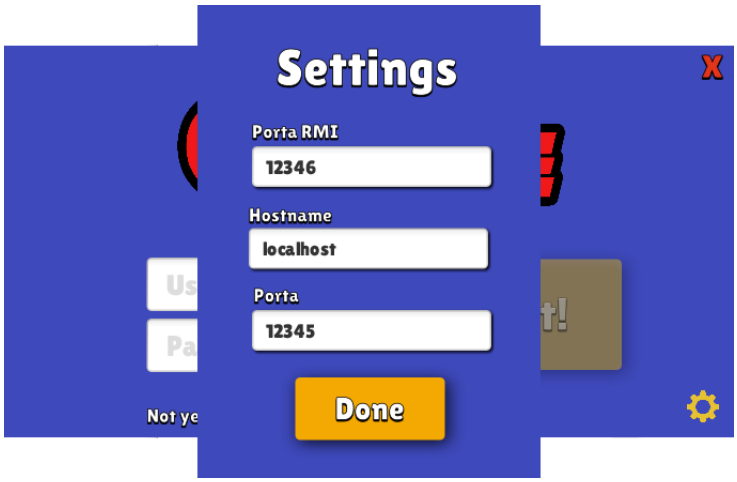
9.1 loginView

9.2 signupView

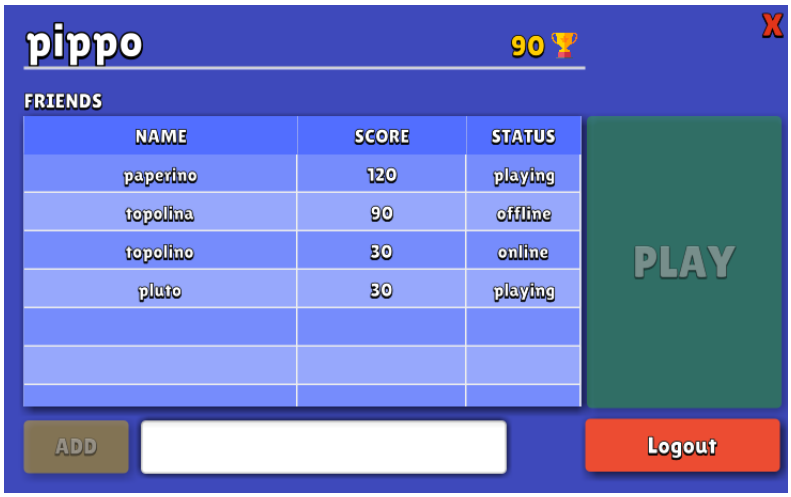




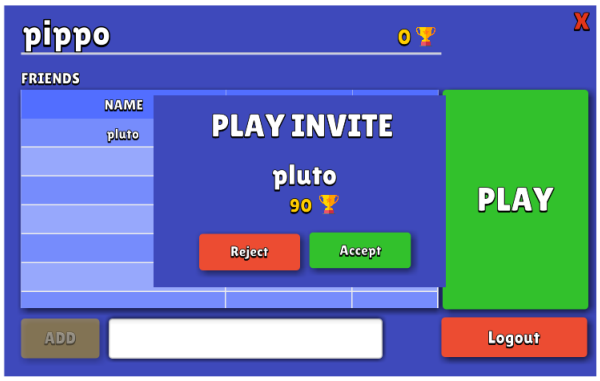
9.3 settingsView



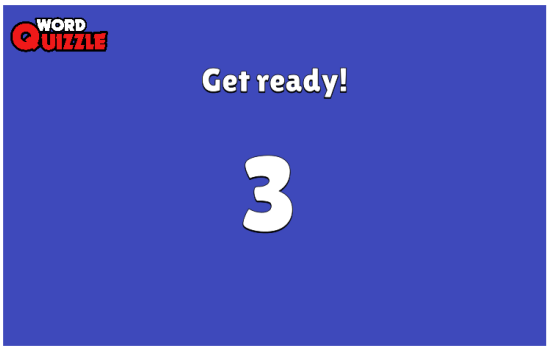
9.4 gameView

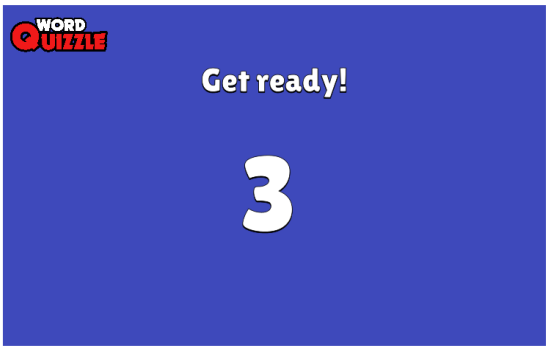


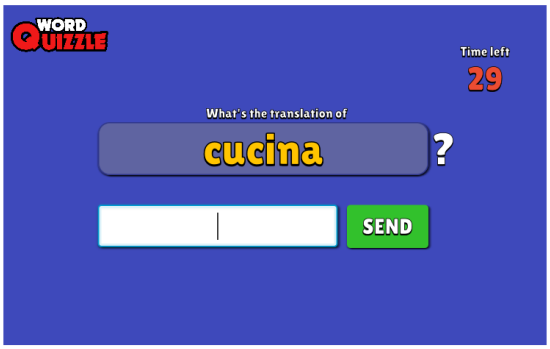
9.5 invitationAlertView e waitingAlertView

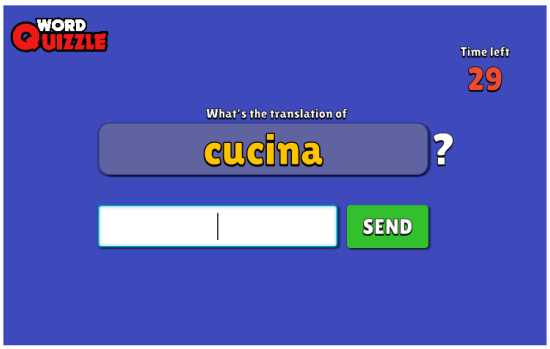


9.6 challengeView

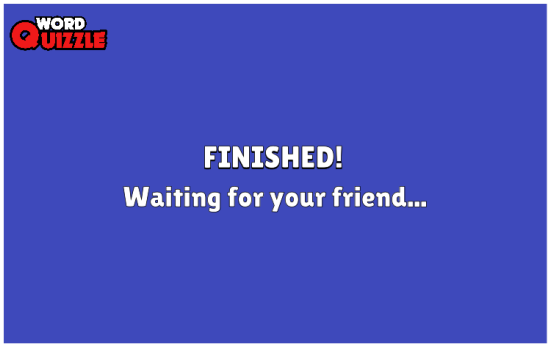


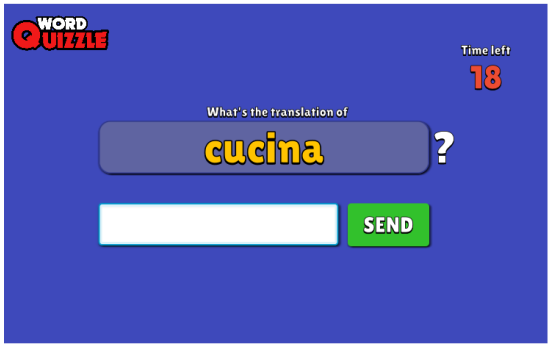






9.7 waitingFriendResultView





9.8 endChallengeView