Documentazione

Lorenzo Iovino

Laboratorio di Reti 2016/2017

Tabella dei contenuti

1

Architettura del sistema

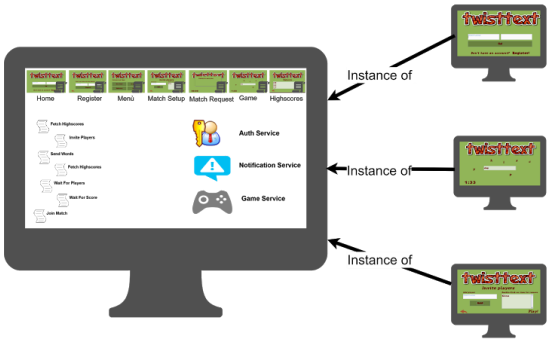
1.1 Overview Generale

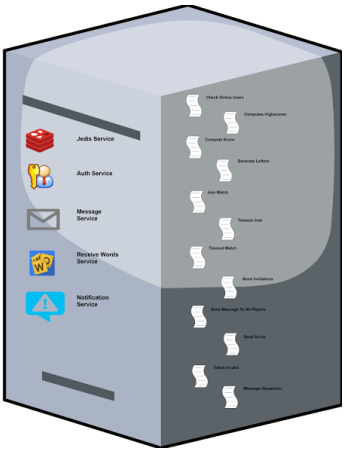
Il sistema consiste di due applicativi software: il **Client** ed il **Server**.

Il Client è di tipo *thin*, quindi per sua natura gestisce solo l'interazione dell'utente con l'interfaccia grafica e richiede i servizi del server attraverso una Application Programming Interface (API).
Il Client è di tipo grafico ed è sviluppato con SWING, inoltre tutte le chiamate alle API sono gestite in modo asincrono per favorire una migliore esperienza utente.

Il Server è un insieme di servizi i quali si occupano di implementare diverse funzionalità per il funzionamento del gioco, la gestione degli utenti e la persistenza. Tutti i servizi sono *threads* indipendenti tra di loro ed ogni servizio può lanciare l'esecuzione di vari *task* i quali sono le unità di calcolo per la realizzazione delle funzionalità demandate dal client.

La API è realizzata utilizzando diverse tecnologie di rete:
RMI per l'autenticazione e il servizio di notifiche;
TCP per lo scambio dei messaggi di gestione del gioco
UDP per l'invio delle parole e dei punteggi





1.2 Message e Response

Le comunicazioni possono essere effettuate utilizzando due diversi modelli per descrivere i messaggi.

I *Message* sono utilizzati per essere inviati su TCP e UDP una volta serializzati.

```
public class Message implements Serializable {
   public String sender;
   public String message;
   public DefaultListModel<String> data;
   public String token;

   public Message(String message, String sender, String token,
   DefaultListModel<String> data) {
       this.message = message;
       this.sender = sender;
       this.data = data;
       this.token = token;
   }
   public static Message toMessage(String data)
   public String toString()
}
```

è possibile convertire dei dati in message e dei dati in stringa, mantenendo un formato standard deciso nell'implementazione del metodo.

Le *Response* sono utilizzati per l' RMI, ed utilizzato i dati strutturati in JSON, evitando conversioni custom come nel caso dei Message.

```
public class Response implements Serializable{
   public String message;
   public Integer code;
   public JsonObject data;

   public Response(String message, Integer code, JsonObject data) {
       this.message = message;
       this.code = code;
       this.data = data;
   }
}
```

Ovviamente entrambi i modelli sono interscambiabili, ma, nello sviluppo dell' applicativo ho preferito inserirli entrambi, utilizzando le *Response* con le RMI, in modo da simulare una risposta API (stile REST) dove è presente un codice di errore, un message e i dati in formato JSON e i *Message* per la comunicazione attraverso socket TCP e UDP di più basso livello.

2

Client

2.1 Il client in breve

Il client è un applicativo per desktop scritto in JAVA di tipo *thin*, in quanto non mantiene in memoria nessun tipo di informazione relativa al gioco se non quelle relative alla sessione attuale.

Le chiamate alle API del client vengono effettuate attraverso degli *swing workers* che permettono di non bloccare l'evoluzione dell'interfaccia grafica nel momento che si rimane in attesa della risposta di una chiamata ad una API, così da risultare non bloccanti e quindi favorire un'esperienza utente più fluida.

Per lo sviluppo del client è stato seguito il pattern di programmazione MVC in modo da separare la logica di presentazione da quella di business.

Nel sorgente i files sono suddivisi in diversi packages:

1) *pages*: implementano la logica di presentazione, concentrandosi sul visualizzare i componenti della UI e gestire le transizioni.
2) *controllers:* forniscono metodi utilizzabili dalla logica di presentazione per richiedere i servizi esposti dai *services*.
3) *services:* si occupano di effettuare le chiamate vere e proprie ai modelli dei dati e al server, in modo da rendere trasparente i dati e i metodi remoti ai *controllers*.
4) *tasks:* vengono chiamati (per lo più dai *services*) quando occorre eseguire operazioni di calcolo asincrono.
5) *ui:* gli elementi della user interface utilizzati per costruire le *pages,* implementati in SWING.

2.2 I servizi

I servizi del client si occupano di creare una interfaccia per i controllers in modo da rendere trasparente all' interfaccia l'elaborazione dei dati e l'esecuzione di servizi sul server.

2.2.1 AuthService

Auth Service è il servizio che si occupa di fornire le funzionalità per il *login*, il *logout* e la *registrazione* degli utenti.
Il servizio, basato su RMI, fornisce una interfaccia *IAuth* sia al client che al server in modo che entrambi possano implementarla.
Il servizio di authentication è esposto tramite registry dal server sull' uri *serverURI/auth.*

2.2.2 GameService

Game Service fornisce le funzionalità per la gestione della sessione di gioco, wrappa il modello e ne espone i metodi.

2.2.3 NotificationClientService

Notification Service il servizio è fornito tramite RMI, l'unico metodo implementato lato Client è il *sendInvite*, che viene eseguito quando arriva l'invito per partecipare ad un match da qualche altro giocatore.
L'arrivo di un nuovo invito chiama la *beginMatch* per dare la possibilità al giocatore di partecipare alla partita.

2.2 I tasks

I tasks sono degli scripts che si occupano di portare a termine un certo obiettivo per poi notificarlo a chi lo ha invocato.

La struttura:

I tasks del client implementano l'interfaccia *SwingWorker* la quale fornisce due metodi che ho utilizzato per simulare un flusso di esecuzione e permettere di concatenare più task tra di loro. Tutti i tasks sono quindi threads.

doInBackgrounds: questo metodo contiene il lavoro da svolgere, che grazie alla natura dello *SwingWorker* verrà svolto in modo asincrono rispetto all'evoluzione dell' interfaccia grafica.

done: viene eseguito alla fine del *doInBackgrounds*, in questo modo alla terminazione del task posso eseguire un'altro task chiamandolo in questa sezione.

N.B. Per poter chiamare dei task in cascata ho passato al costruttore del task (dove necessario) una callback di tipo *Callable* o *SwingWorker* che ho poi richiamato al *done,* simulando un approccio funzionale.

2.3.1 InvitePlayers

Invita i giocatori del match creato spedendo un Message tramite TCP con:

```
message: "START_GAME"
sender: String usernameDelCreatoreDelMatch
token: String tokenDelCreatoreDelMAtch
data: DefaultListModel<String> listaDegliUsernameDegliInvitati
```

2.3.2 WaitForPlayers

Mette il gioco in attesa e attende che tutti i giocatori joinino.
Nell'attesa attiva legge il socket TCP aspettando un Message di tipo:

JOIN_TIMEOUT: Si è verificato il timeout, annulla il match corrente e torna al menù
MATCH_NOT_AVAILABLE: Il match a cui si vuole partecipare non è più disponibile a causa di un timeout.
GAME_STARTED: Il gioco è pronto per iniziare, recupero le lettere e mi registro al multicast per ricevere i punteggi alla fine.

Se il Message contiene un messaggio uguale a GAME_STARTED la callback che notifica all'utente che il gioco è pronto viene eseguita.

2.3.3 SendWords

Invia le parole inserite durante la sessione di gioco spedendo un Message tramite UDP con:

```
message: "WORDS"
sender: String usernameDelCreatoreDelMatch
token: String tokenDelCreatoreDelMAtch
data: DefaultListModel<String> listaDelleParole
```

Alla fine richiama la callback WaitForScore.

2.3.4 WaitForScore

Resta in attesa di ricevere tramite Multicast UDP i punteggi finali.
Il Message da ricevere ha il messaggio uguale a FINALSCORE, che conterrà nella sezione *data* i punteggi del match appena concluso.
Infine chiama la callback che conclude il gioco e visualizza la HighscoresPage (riutilizzo dello stesso componente degli highscore per visualizzare i punteggi parziali del match)

2.3.5 FetchHighscores

Invia un Message tramite TCP per richiedere gli highscores con:

```
message: "FETCH_HIGHSCORES"
sender: String usernameDelCreatoreDelMatch
token: String tokenDelCreatoreDelMAtch
data: DefaultListModel<String> []
```

e attende la risposta di un Message con un messaggio uguale a "HIGHSCORES" che conterrà nella sezione *data* i punteggi globali.
Infine chiama la callback che visualizza la HighscoresPage.

2.3.6 BeginMatch

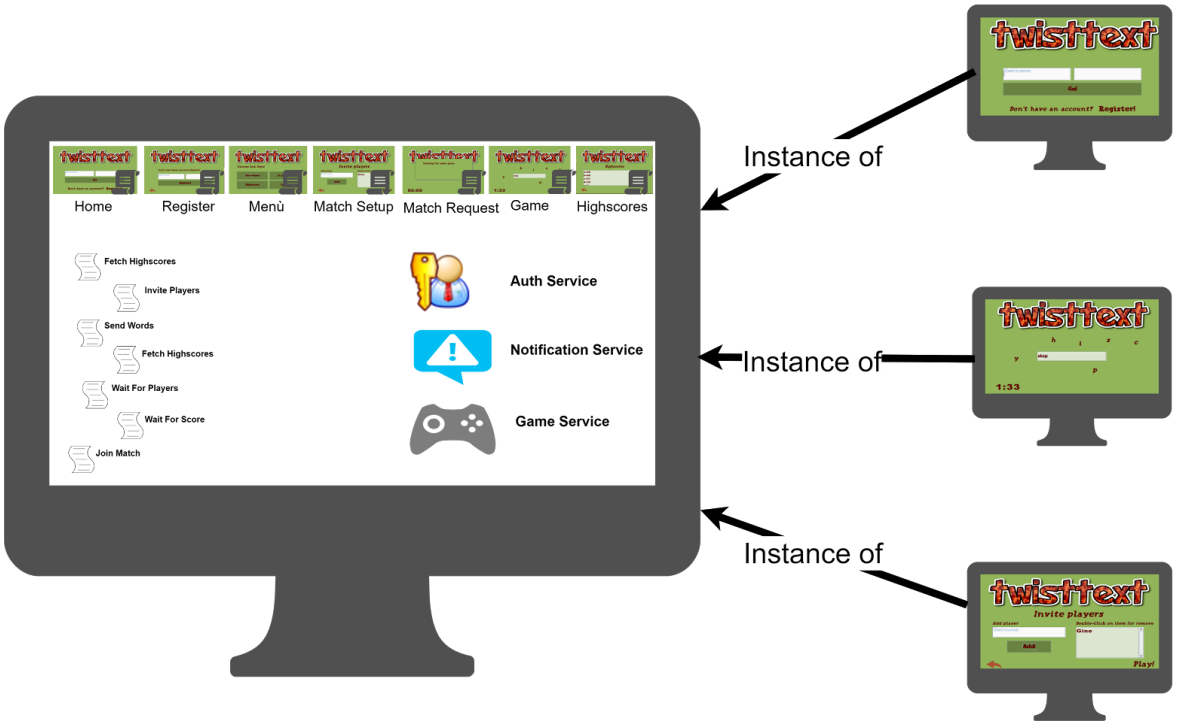
Viene eseguito quando si riceve una notifica di un nuovo match.
Il task aggiunge il match alla pendingList e visualizza il dialog che permette di joinare.

2.3.7 JoinMatch

Il task procede all' invio di un Message tramite TCP con messaggio uguale a JOIN_MATCH con:

```
message: “JOIN_GAME”
sender: String usernameDelCreatoreDelMatch
token: String tokenDelCreatoreDelMAtch
data: DefaultListModel<String> IlNomeDelMatchDaJoinare
```

N.B. Il nome di un match da joinare corrisponde al nome utente del suo creatore.



3

Server

3.1 Il server in breve

Il server è un applicativo console scritto in JAVA, esso definisce le signature delle API esponendo i servizi, si occupa di gestire le sessioni di gioco, mantiene lo stato del sistema e ha un meccanismo per salvare alcune informazioni sul DB.

Il server alla sua esecuzione fa partire dei servizi (Services) ed occupa alcune porte.

Nome: AuthService	*Porta: 9999*
Nome: JedisService	*Porta: 6379*
Nome: MessageService	*Porta: 10000*
Nome: NotificationService	*Porta: 20000*
Nome: ReceiveWordsService	*Porta: 10001*

Tutti i servizi sono Thread figli del processo Server e dipendono dal suo stato di esecuzione..

Nel sorgente i files sono suddivisi in diversi packages:

1) *models:* rappresentano i modelli dei dati e forniscono i metodi per cambiarne e/o ottenerne lo stato. La rappresentazione dei modelli usa strutture synchronized, quindi è thread safe.
2) *services:* i servizi dell'applicazione, tutti runnati come Thread separati e indipendenti. Essi implementano diverse funzionalità.
3) *tasks:* come per il client, sono dei thread "usa e getta", atti ad essere utilizzati per eseguire calcoli e/o operazioni asincrone.

3.2 I servizi

I servizi del server implementano l'interfaccia API esposta ai client e alcuni meccanismi richiamati dall'interno. Sono lanciati con thread separati, quindi non dipendono l'uno dall'altro, questo significa che se un servizio si blocca gli altri continuano a funzionare.

3.2.1 AuthService

Auth Service è il servizio che si occupa di fornire le funzionalità per il *login*, il *logout* e la *registrazione* degli utenti.
Le sessioni valide vengono verificate con l'utilizzo di un token, inviato all'utente come risposta del login (se effettuato con successo), il token deve essere inviato ad ogni successiva richiesta.
Il servizio è esposto tramite registry dal server sull' uri *serverURI/auth.*

3.2.2 ReceiveWordsService

Receive Words Service il servizio resta in attesa di ricevere delle parole dai client. Una volta ricevute delle parole, delega al task ComputeScore il calcolo del valore.

3.2.3 MessageService

Message Service resta in attesa di ricevere messaggi sul socket TCP e delegarne il dispatchamento al task MessageDispatcher, che ne valuterà la natura e deciderà l'azione da eseguire.

3.2.4 NotificationService

Notification Service il servizio è basato su RMI e permette di inviare le notifiche ai clients che si sono registrati alla callback, invocando il metodo remoto fornito dagli stessi clients (*sendInvite*).

3.2.5 JedisService

Jedis Service

è un servizio che si interfaccia con Redis (in-memory db) il quale è in esecuzione sull'host (seguire i passi per l'installazione nella sezione 4).
Permette di salvare in memoria delle tuple <key,value> e successivamente accedervi. Viene utilizzato per mantenere lo stato del Server ad ogni sua ri-esecuzione, ma, se si vuole ottenere una persistenza dei dati anche allo spegnimento della macchina host, occorre utilizzare un DB, mantenendo Redis come cache.

Note:
Ci sono molti DB che si interfacciano direttamente con Redis, quindi è uno sviluppo che si può realizzare anche senza modificare il codice del Server, a piacere dell'amministratore della macchina host.
La chiave dove Redis memorizza i dati è "users" basterà effettuare un binding tra quella chiave e una tabella del DB. Consiglio un db non relazionale (Es: MongoDB) per questo sviluppo futuro.

3.2 I tasks

Come per il client i tasks sono degli scripts che si occupano di portare a termine un certo obiettivo per poi notificarlo a chi lo ha invocato.

La struttura:

I tasks del server implementano l'interfaccia *Callable* la quale fornisce il metodo *call()* che conterrà le istruzioni per svolgere le operazioni richieste.

Alcuni task vengono lanciati e producono dei *side effects* su alcune proprietà synchronized dell'istanza di un certo *match*, questo garantisce che l'algoritmo eseguito sia thread safe nonostante il flusso spezzato dai *side effect.*
Altri task invece aspettano un valore di risposta risultando bloccanti rispetto al flusso (utilizzando la *get()* della *Callable*).

La scelta nell'utilizzo di uno piuttosto che dell'altro approccio è guidata dal singolo caso di utilizzo, in quanto a volte è necessario aspettare un certo valore o collezionare più valori prima di proseguire con il flusso (Es: *joinMatch*) e in altri casi è una semplice esecuzione di un task a bassa priorità, il quale non importa che finisca in un istante ben preciso.

3.3.1 CheckOnlineUsers

Verifica che un utente sia presente tra gli utenti connessi (modello *Sessions*), utilizzando l'username come chiave per la ricerca.

Valore di ritorno *true/false* per l'attesa nel flusso e nessun side effect.

3.3.2 SendInvitations

Invia una notifica utilizzando il servizio NotificationService a tutti gli utenti che partecipano al match.

Valore di ritorno *true/false* per l'attesa nel flusso e nessun side effect.

3.3.3 SendMessageToAllPlayers

Invia un messaggio di tipo Message tramite TCP a tutti gli utenti di un match (utilizzato per inviare messaggi di servizio come per esempio il JOIN_TIMEOUT).

Valore di ritorno *true/false* per l'attesa nel flusso e nessun side effect.

3.3.4 SendFinalScores

Invia i punteggi finali in multicast a tutti i giocatori iscritti al gruppo multicast del match in questione, quindi rimuove il match dai matches attivi (proprietà *activeMatches* di match).

Nessun valore di ritorno e nessun side effect.

3.3.5 JoinMatch

Modifica il match creato dal giocatore specificato nel Message alla voce *sender* aggiornando il *playerStatus* del giocatore che joina ad 1 (0=non partecipa al match, 1=partecipa al match) e il *playerSocket* con il valore del socket creato per la comunicazione con il giocatore.

Valore di ritorno *true/false* per l'attesa nel flusso, come side effect invece viene settato il valore di *joinTimeout a false* solo se tutti i giocatori hanno correttamente joinato al match, quindi prevenendo l'esecuzione del timeout con la conseguenza di cancellare il match.

3.3.6 TimeoutJoin

Setta il valore di *joinTimeout* del match in questione a *true* e si mette in attesa il Thread per 7 minuti.

Allo scadere dei 7 minuti si riverifica il valore di *joinTimeout,* se è rimasto *true,* allora il timeout ha effetto, se invece è *false* (modificato dalla JoinMatch perchè tutti i giocatori hanno joinato correttamente) si ignora il timeout.

Valore di ritorno *true/false* per l'attesa nel flusso, modifica della proprietà *joinTimeout* del match come side effect.

3.3.7 TimeoutMatch

Quando il match inizia, viene inizializzato anche questo task, che resta in attesa per 5 minuti. Allo scadere del tempo si verifica se la proprietà *matchTimeout* del match in questione è *true* e nel caso settare a 0 i punteggi dei giocatori che non hanno ancora inviato le parole, quindi procedere con la terminazione del match inviando i finalScores.

La proprietà *matchTimeout* viene modificata a *false* quando tutti i giocatori hanno inviato le loro parole.

Valore di ritorno *true/false*, ma nessuna attesa nel flusso perché viene lanciato il task *SendFinalScore*.

3.3.8 GenerateLetters

Sceglie una parola a caso dal dizionario e ne restituisce la lista delle lettere

Valore di ritorno *DefaultListModel*<*String*> cioè la lista delle lettere e nessun side effect.

3.3.9 ComputeScore

Calcola il punteggio in base alle parole inviate.
Il punteggio è uguale all lunghezza della parola e solo se ritenuta valida, cioè presente nel dizionario e non doppione.
Setta il punteggio dell'utente con quello calcolato e se è l'ultimo utente che invia le lettere, allora termina il match settando la proprietà *matchTimeout* di match a *false,* settando a 0 il punteggio degli utenti che hanno inviato una lista di parole vuota e chiamando il task *SendFinalScore.*

Valore di ritorno *score* ma non utilizzato e nessun side effect.

3.3.10 ComputeHighscores

Legge i punteggi dei giocatori e li formatta come stringhe “username:score” per l’invio in un *Message* tramite TCP con messaggio uguale a HIGHSCORES.

Valore di ritorno *DefaultListModel*<*String*> cioè la lista degli highscores e nessun side effect.

3.3.11 TokenInvalid

Invia il Message tramite TCP con messaggio uguale a TOKEN_NOT_VALID.

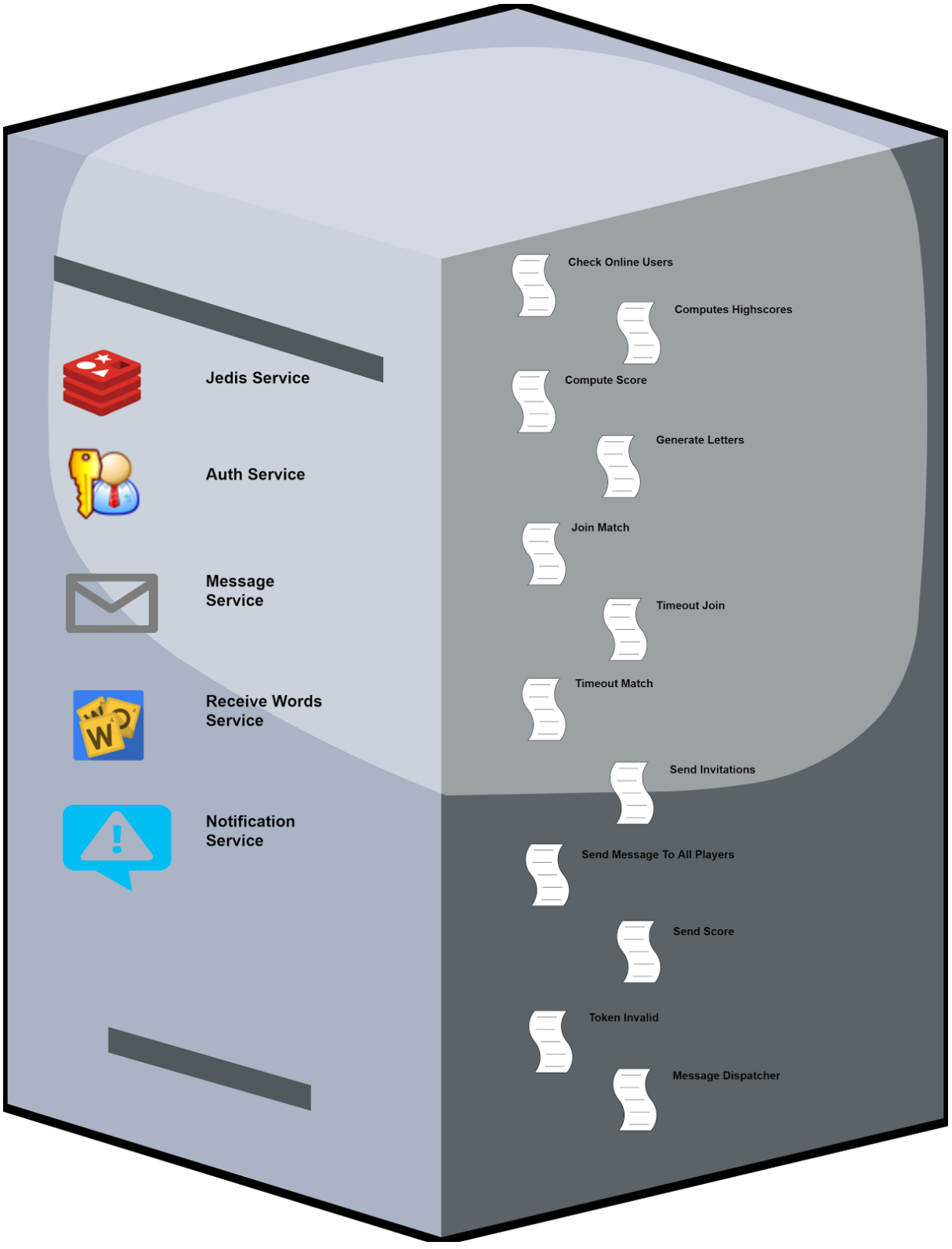
Nessun valore di ritorno e nessun side effect.

3.3.12 MessageDispatcher

Valuta il messaggio di un Message ricevuto come parametro ed esegue l’azione appropriata, selezionando tra una delle azioni iniziali possibili nel flusso dei messaggi TCP.

I messaggi iniziali sono: START_GAME, FETCH_HIGHSCORES e JOIN_GAME.

Valore di ritorno *true/false* anche se non utilizzati e nessun side effect.


Jedis Service
Auth Service
Message Service
Receive Words Service
Notification Service
Check Online Users
Computes Highscores
Compute Score
Generate Letters
Join Match
Timeout Join
Timeout Match
Send Invitations
Send Message To All Players
Send Score
Token Invalid
Message Dispatcher

4

Installazione

4.1 Installazione degli applicativi

Prerequisiti software:

1) Java
    a) Scaricare JAVA dal sito https://www.java.com/it/download/
    b) Installare il pacchetto scaricato
2) Redis
    a) Scaricare REDIS dal sito https://redis.io/download
    b) Installare il pacchetto scaricato
    c) Abilitare REDIS come servizio:
        i) Su Windows 10:
            (1) Premere Start
            (2) Digitare “Servizi” e cliccare sull’icona “Servizi”
            (3) Cercare nella lista il servizio “Redis Service”, cliccare con il tasto destro del mouse e selezionare dal menù a tendina la voce “Attiva”.
        ii) Su Linux:
            (1) Aprire un terminale ed eseguire: `redis-server`
            (2) Verificare che redis sia partito digitando: `redis-cli ping`

4.2 Esecuzione

Esecuzione da Desktop Environment

Su windows:

I file distribuiti sono dei file jar, quindi su windows autoeseguibili cliccandoci sopra:

IMPORTANTE eseguire il server per primo!

1) Eseguire il server cliccando su *Server_jar/Server.jar*
2) Eseguire quanti client si vogliono cliccando su *Client_jar/Client.jar*

Esecuzione da Terminale

Su windows e su linux:

IMPORTANTE eseguire il server per primo!

1) Aprire un terminale
2) Portarsi nella cartella dell' applicativo *Server_jar* ed eseguire

```
java -jar Server.jar
```

3) Portarsi nella cartella dell' applicativo *Client_jar* ed eseguire

```
java -jar Client.jar
```

Nota:

Se il server non si esegue probabilmente è perchè qualche porta richiesta dal server è già utilizzata da qualche altro servizio o da un'istanza del server stesso.
Per fixare quest'ultimo problema basta eseguire:

Su windows:

1) Ctrl-SHIFT-ESC
2) Cercare il processo del server dalla lista (ha l'icona di java)
3) Terminarlo

Su linux:

1) Aprire un terminale ed eseguire

```
sudo kill -9 $(ps -a | grep Server | grep -v grep | awk '{print &1}')
```