Laboratorio di Reti

Anno Accademico 2016/2017

Text-Twist

A cura di
Gemma Martini

17 aprile 2017

Indice

1 Introduzione

Questo progetto è stato realizzato con uno stile di programmazione difensiva e con particolare attenzione alla gestione della concorrenza. L'interfaccia è di tipo testuale, per semplicità, e ha lo standard di stampare i messaggi di errore in rosso, mentre tutti gli altri sono stampati in nero.

2 Overview del programma

Il *software* da realizzare ha richiesto l'implementazione di tre diversi protocolli di comunicazione: RMI, TCP e UDP. Gli ultimi due sono realizzati mediante *thread*, che si occupano di gestire le varia funzionalità del programma: rispettivamente richiesta nuove partite, risposta agli inviti pendenti, richiesta delle partite e invio parole al *server* e invio classifica della partita. Il primo, invece, non necessita della creazione di un *thread* esplicito, poichè l'implementazione di Java ne fornisce uno di default. È necessario premettere che, per l'utilizzo della piattaforma con i *client* che non si trovano sullo stesso computer del *server*, si deve specificare l'"textithost del server da linea di comando (argomento della funzione `main` del *client*. Il protocollo TCP è implementato mediante lo scambio di messaggi, che sono modellati mediante l'omonima classe `Message`, specializzata dalle sottoclassi, che contengono campi specifici per il tipo di messaggio.

3 Classi Java

Il codice è stutturato in tre pacchetti: uno esclusivo del *client*, uno esclusivo del *server* ed uno condiviso.

In allegato il file `Doxygen` del progetto.

Sempre in allegato si trova una versione PDF del codice del progetto.

In figura 1 si può trovare il diagramma delle classi.

3.1 Lato server

3.1.1 `EditThread`

Campi

- `private Server TextTwistServer;`
- `private Path rankFilePath;`
- `public ArrayBlockingQueue<Integer> whenToAct;`

Costruttore, che assegna i valori di: capacità massima, puntatore al server, path del file classifica.

fromHashMapToFile, che aggiorna il file della classifica con l'hashmap attuale. Come primo passo, crea un file nuovo con i dati aggiornati e, come secondo passo, lo sostituisce a quello vecchio. È stato scelto di creare sempre un nuovo file e di sovrascrivere quello non aggiornato, così da evitare errori dovuti alla concorrenza.

RUN, che:

1. Prova ad estrarre un elemento dalla coda. Se è vuota si blocca in attesa.
2. Se riesce a prendere un elemento, provvede alla sovrascrittura del file.

3.1.2 `Server`

Fatta dalle seguenti parti:

CAMPI

- `public int wordLength = 7;`
- `public int editThreadCapacity = 10000;`
- `private HashMap<String, User> usernameAndUser;`
- `private Set<String> onlineUsers;`
- `private HashMap<Integer, AnsweringMatch> allAboutPendingInvitations;`
- `private Random random;`
- `public ArrayList<String> possibleMatchWords;`
- `public Set<String> dictionary;`
- `public EditThread editThread;`
- `public TCPThread tCPThread;`
- `public UDPThread uDPThread;`
- `public Integer IDEvaluationSeed;`
- `public Integer IDInvitationSeed;`
- `public Registry serverRegistry;`
- `public Path dictionaryPathName = Paths.get("dictionary.txt");`
- `public Path rankPathName = Paths.get("score.txt");`

COSTRUTTORE, che alloca la memoria per le variabili di istanza non primitive e, dati i file di classifica e dizionario, li trasforma rispettivamente in *Hash Map* e *Set* e li memorizza nella struttura dati.

GENERAZIONE DI UN INTERO CASUALE, che ritorna un intero da 1 ad un massimo.

INSERIMENTO, REPERIMENTO E CANCELLAZIONE DI UNA PARTITA, che sono tre diverse funzioni, che effettuano queste tre operazioni gestendo acquisendo un lock sulla struttura che memorizza le informazioni, in modo da evitare problemi di concorrenza.

INSERIMENTO, CANCELLAZIONE E CHECK DI UTENTI IN LINEA, analoghe al punto precedente.

INSERIMENTO, REPERIMENTO SINGOLO E GLOBALE DI UTENTI, analoghe al punto precedente. Il reperimento di tutti gli utenti crea una nuova lista che li contiene tutti, in modo da consentire l'iterazione senza problemi di modifiche concorrenti.

Ordinamento degli utenti in base ai punteggi, che ritorna una lista ordinata di utenti.

Login, funzione remota per il client, che:

1. Controlla che esista lo username;
2. Controlla che la password corrisponda;
3. Aggiunge lo username ottenuto nel set OnlineUsers di Server.

Registrazione, funzione remota per il client, che:

1. Controlla atomicamente che l'username non sia già esistente e, in tal caso, crea l'utente;
2. Segnala a `EditThread` di aggiornare la versione su disco della lista utenti.

Logout, funzione remota per il client, che rimuove l'utente dal *set* `OnlineUsers`.

Aggiornamento punteggio, che gestisce la modifica del punteggio di un utente segnala a `EditThread` di aggiornare la versione su disco della lista utenti.

Main, che:

1. Crea il server;
2. Crea il registro RMI e vi pubblica la propria interfaccia;
3. Avvia il thread che si occupa del salvataggio su disco dei cambiamenti;
4. Avvia il thread che si occupa della gestione delle richieste TCP;
5. Avvia il thread che si occupa della gestione delle richieste UDP;

3.1.3 `TCPThread`

Campi

- `private int port = 6789;`
- `Server TextTwistServer;`

Costruttore, che assegna la struttura dati del server.

Apertura connessione TCP, che crea un socket e si mette in ascolto, stampando un messaggio di successo.

Chiusura connessione TCP, che chiude il socket e stampa un messaggio di successo.

Task per la thread pool, che riceve il socket di comunicazione, stampa un messaggio di successo e si occupa di gestire il client:

1. Prova a leggere dal socket: se è vuoto aspetta
2. Gestisce il messaggio ricevuto:

- "invitation" → crea la struttura che rappresenta il match, dopodichè se i partecipanti sono tutti online manda a tutti l'invito via RMI. Fatto questo, avvia un timer di 7 minuti e si mette in attesa di ricevere la risposta a tutti gli inviti. Se la partita parte con successo, avvia un ulteriore timer di 5 minuti e si mette in attesa di ricevere le liste di parole dai partecipanti.
- "invitation reply" → se la risposta è affermativa allora memorizza questo fatto e il socket su cui è avvenuta la comunicazione, in modo da poter successivamente confermare l'avvio della partita o segnalare un errore. Altrimenti chiude la comunicazione senza fare altro.
- "rank" → genera e invia la classifica.

Run, che:

1. Stampa un messaggio all'avvio;
2. Crea una thread pool e stampa un messaggio;
3. Apre un socket in ascolto e, appena qualcuno si connette, passa il socket di comunicazione al worker thread della threadpool.

3.1.4 `UDPThread`

Fatta delle seguenti parti:

Campi
- `private final int port = 6790;`
- `private final Server textTwistServer;`
- `private final DatagramSocket socket;`

Costruttore, inizializza il riferimento al server e crea il socket per l'ascolto

Conta caratteri, una funzione per contare il numero di occorrenze di un carattere in una stringa.

Controlla anagramma, una funzione che controlla se una certa parola può essere legittimamente ottenuta a partire da un'altra.

Run, la funzione che riceve messaggi via UDP, ne estrae la lista delle parole inviata dall'utente e assegna il punteggio.

3.1.5 `User`

Fatta delle seguenti parti:

Campi
- `public final String username;`
- `public final String password;`
- `private int score;`

Aggiornamento, ottenimento e confronto del punteggio, che avvengono acquisendo un lock in modo da evitare problemi legati alla concorrenza.

3.1.6 `Match`

Fatta delle seguenti parti:

Campi

- `public final Integer invitationID;`
- `public final String *matchWord;`
- `private final Server textTwistServer;`
- `private HashMap<String, Boolean> pendingInvitation;`
- `private HashMap<String, Socket> usernameAndReplySocket;`
- `public ArrayList<UsernameAndScore> usernameAndScore;`

Inserimento e iterazione sui socket degli utenti, sincronizzate con i lock nativi di Java.

Modifica e ottenimento dello stato dell'invito, anche queste sincronizzate con i lock nativi di Java.

Anagramma della parola , che ottiene un anagramma casuale della parola del match.

Indirizzo del match , che ottiene un indirizzo multicast univoco associato alla partita.

Attesa e risposta ai client, due funzioni che si occupano di verificare quando il server è in grado di rispondere ai client, inviandogli o la parola che rappresenta il match via TCP (o un errore) oppure la classifica in multicast.

3.2 Lato client

3.2.1 `Client`

Campi

- `public boolean quit;`
- `private HashMap<Integer, ClientSideMatch> pendingMatches;`
- `public User whoIAm;`
- `private static Registry serverRegistry;`
- `private String host;`
- `private static final int tCPportNumber = 6789;`
- `public static final int uDPportNumber = 6790;`
- `public static final int multicastportNumber = 6791;`

Costruttore, che assegna false ad isLogged, true a matchEnded, false a quit, alloca la memoria per pendingMatches e assegna whoIAm a null;

Visualizzazione opzioni da sloggato : ritorna l'oggetto utente, che ha effettuato login. In caso di errore o di registrazione ritorna null.

- Login → controlla la non nullità delle stringhe e poi effettua procedura RMI, cambiando il valore di isLogged. Inoltre registra se stesso nel server RMI;
- Registrazione → controlla la non nullità delle stringhe e poi effettua la procedura RMI;

Funzione di visualizzazione opzioni da loggato : crea un socket per comunicare con il server.

- `Partita` → chiede numero e nomi degli avversari, per poi leggere gli username (tutti distinti) e mandarli con TCP al server. Arriva un invito anche all'utente stesso;
- `Richieste Pendenti` → se ne esiste almeno una, chiede all'utente di selezionare una richiesta a cui rispondere, per poi mandare la risposta con TCP al server. Nel caso in cui la risposta sia no, questa viene prima eliminata dalla lista delle richieste. Nel caso in cui la risposta sia sì viene eliminata l'intera lista. Rimane in attesa fino all'avvio della partita o al suo annullamento;
- `Classifica` -> manda la richiesta TCP al server;
- `Logout` -> manda la richiesta RMI al server, cambiando il valore di isLogged;

Callback per server : stampa a video un avviso all'utente ed aggiunge un elemento alla lista delle richieste pendenti.

Gestione creazione partita : gestisce la creazione della partita e attende la risposta del server.

Gestione partita : legge le risposte dell'utente, le invia al server via UDP alla fine del timeout, dopodichè attende (fino a 5 minuti) di ricevere la classifica via multicast e la stampa.

Invio messaggio : invia un messaggio al server e, opzionalmente, restituisce la risposta ricevuta.

Main, che:

1. Inizializza il client;
2. Memorizza lo stub del server;
3. Cicla, con le opportune condizioni, facendo alternare schermata da sloggato e schermata da loggato;
4. Stampa un messaggio di benvenuto;
5. A seconda di cosa ha selezionato l'utente sloggato può:
   - Effettuare login (via RMI);
   - Registrarsi (via RMI).
6. A seconda di cosa ha selezionato l'utente loggato può:
   - Richiedere l'inizio di una partita al server (via TCP);
   - Visualizzare le richieste pendenti;
   - Visualizzare la classifica richiedendolo al server (via TCP);
   - Effettuare il logout (via RMI).

3.2.2 `ClientSideMatch`

Fatta delle seguenti parti:

CAMPI
- `public final String inviterUsername;`
- `private final long creationInstant;`
- `private final long howOldICanBe = 7*60*1000;`

GESTIONE SCADENZA, permette di capire quando una richiesta memorizzata dal client è ormai scaduta per il server.

3.3 Condivise

La descrizione delle classi mancanti è abbastanza semplice e può essere trovata nel manuale generato con doxygen.

3.3.1 `IO`

Contiene le seguenti funzioni:

`out` Scrive su standard output, serializzando gli accessi con un lock globale.

`err` Scrive su standard error, serializzando gli accessi un lock globale.

Si è scelto di utilizzare, per la stampa su standard output/error, questa classe, di modo che sia possibile, nel caso si vogliano avere più righe di output consecutive, acquisire semplicemente il lock su `IO.class`.

3.3.2 `CounterThread`

Fatta delle seguenti parti:

CAMPI
- `public volatile boolean ended = false;`
- `private final int minutes;`
- `private final String job;`

RUN, attende il numero di minuti specificato e, in base al job, eventualmente stampa un messaggio.

4 Esecuzione

Il programma può essere eseguito avviando prima il lato server, nella cartella in cui si trova il dizionario (chiamato `dictionary.txt`):

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

e poi i vari client per i giocatori:

```
java -jar Client.jar [hostname]
```

dove `hostname` rappresenta l'hostname o l'indirizzo IP del computer su cui si trova il lato server.

5 Scelte implementative

Si è scelto di gestire la concorrenza sulle variabili rendendole private e fornendo metodi per accedervi che siano *thread-safe*. Inoltre, per alcuni campi ciò non si è reso necessario, perchè `final` oppure perchè vi accede un solo thread.

La gestione dei timer nelle varie fasi di gioco è affidata ad un thread che aspetta per il tempo necessario e, successivamente, imposta un flag a `true`. Questa gestione dei timer porta con sè un errore dell'ordine dell'intervallo di tempo che intercorre tra un controllo del flag e il successivo, nella fattispecie mezzo secondo (contro timer dell'ordine dei minuti).

Infine, la scrittura persistente dei dati è affidata tutta a un singolo thread, in modo da evitare problemi di scritture contemporanee. Inoltre, per avere maggior robustezza, la scrittura avviene su un file temporaneo che va poi a sovrascrivere il file già esistente: in tal modo, se dovesse avvenire un crash durante la scrittura, non si perdono i dati precedenti.

6 Conclusioni

Si conclude che il programma si confà a tutte le richieste del committente e, tramite un'interfaccia snella e pulita, guida l'utente attraverso il gioco. Da notare che la concorrenza di *input* ed *output* è stata gestita con attenzione, ma a causa della mancata corrispondenza tra l'istante in cui viene fatta richiesta di stampa alla *console* e l'istante in cui essa stampa a video, talvolta gli output risultano in ordine sbagliato.

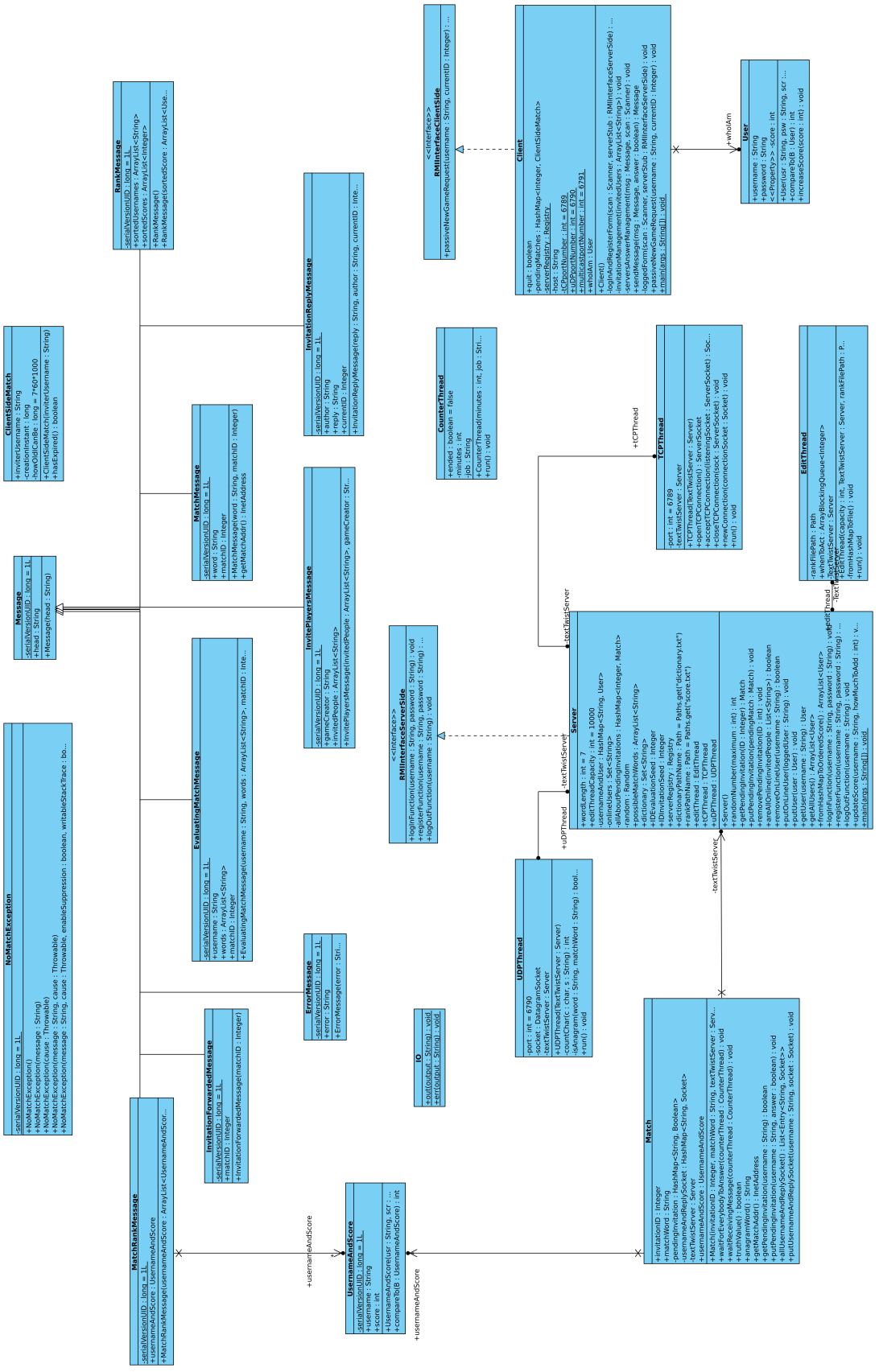

Figura 1: Diagramma delle classi