Relazione

Federico Bernacca, 536683

Indice

1 Introduzione

Il progetto, volto a simulare un sistema Client-Server secondo le direttive assegnate, è stato realizzato mediante le scelte implementative che verranno illustrate a seguire.

1.1 Server

È stata presa la decisione di implementare il server utilizzando un'architettura **NIO**, e sfruttando la potenza dei selettori e dei canali non bloccanti ad esso registrati.
Decisione presa per diverse ragioni:

- Risparmiare il costo di thread switching: realizzando il server in versione multithreaded, si sarebbe ottenuto un thread per client. Il passaggio da un thread all'altro è costoso per un sistema operativo, ogni thread occupa memoria, e si ha un numero limitato di thread creabili.
- Velocizzare i tempi: in una architettura Client-Server come *WordQuizzle*, dove si hanno migliaia di connessioni aperte contemporaneamente che inviano e ricevono solo pochi dati, **NIO** è probabilmente un vantaggio, in quanto la singola interazione richiesta/risposta è molto rapida.
- Gestire in maniera più semplice la concorrenza: non essendoci un vero e proprio parallelismo per la maggior parte delle operazioni, si ha meno bisogno di esplicitare la sincronizzazione delle strutture dati.

1.2 Client

Il client è invece stato implementato usando il classico **IO** in quanto, non avendo bisogno di un selettore o canali bloccanti, l'implementazione **NIO** non avrebbe apportato un grosso vantaggio.

1.3 Files

Tutte le interazioni con i files utilizzano **NIO** coi rispettivi ***FileChannel***. Il file contenente le parole da tradurre, e quello contenente gli utenti e le rispettive informazioni, potrebbero essere molto grandi. Un approccio **NIO** ne aumenta le performance di lettura e scrittura.

1.4 Serializzazione

I dati relativi agli utenti registrati sono salvati sul file *users.json*.
Per serializzare e deserializzare è stata usata la libreria ***Gson*** di Google.
La serializzazione avviene al termine di ogni operazione sensibile:

1. Registrazione di un utente.

2. Creazione relazione di amicizia.

3. Aggiornamento del punteggio dei due utenti a fine sfida.

L'obiettivo è quello di salvaguardare il più possibile i dati in caso di crash improvviso del server. La deserializzazione avviene durante la fase di setup del Server.

1.5 Suddivisione in packages

Avendo implementato diverse classi, per una corretta gestione sono stati creati diversi packages allo scopo di mantenere un certo ordine.

1. **commons**: alcuni parametri e metodi sono comuni a più classi e si trovano in **UtilityClass.java**. Essa contiene solo metodi statici ed è dichiarata *final* in modo che non sia estendibile e presenta un costruttore privato per far sì che la classe non sia istanziabile.

2. **server**: contenente tutte le classi necessarie al Server per una corretta istanziazione.

3. **client**: contenente tutte le classi necessarie al Client per una corretta istanziazione.

4. **cli**: contenente tutte le classi necessarie per eseguire il test con Command Line Interface.

5. **gui**: contenente tutte le classi necessarie per eseguire il test con Graphical User Interface.

La descrizione delle relative classi verrà discussa nel prossimi paragrafi.

2 Server

2.1 Descrizione generale

La classe **Server.java** si trova nel package **server** e presenta al suo interno le seguenti varibili di istanza:

```
1     // canale del server
2     private ServerSocketChannel serverSocketChannel;
3     // selettore che seleziona i canali pronti
4     private Selector selector;
5     // serializza e deserializza
6     private Gson gson;
7     // gestisce richieste di sfida
8     private ThreadPoolExecutor requests;
9     // gestisce le sfide
10    private ThreadPoolExecutor games;
11    // utenti registrati
12    private ConcurrentHashMap<String, User> registeredUsers;
13    // utenti connessi
14    private HashMap<String, UserItem> connectedUsers;
15    // parole italiane estratte durante la configurazione del
      server
16    private String[] italianWords;
```

Listing 1: Server

All'avvio viene creato il ***serverSocketChannel*** ed aperto il ***selector***, col quale si registra il server con annessa operazione di accettazione connessioni. Successivamente, viene avviato il task atto alla registrazione di utenti tramite RMI.
Inizia poi il ciclo nel quale il server attende le richieste dai vari client. Quando un client richiede la connessione, viene registrato anche il suo canale col selettore, ed è a questo punto che inizia la sessione.

2.2 Threads

Il server, una volta avviato e configurato correttamente, crea due ThreadPool.

1. ***ThreadPoolExecutor requests***: adibito all'inoltro di richieste di sfida tramite datagrammi UDP, da un utente X ad un utente Y.

2. ***ThreadPoolExecutor games***: adibito alla gestione delle sfide tra gli utenti.

2.2.1 Richieste di sfida

Quando l'utente X richiede l'inoltro di richiesta di sfida all'utente Y al server, quest'ultimo si accerta che Y sia suo amico e che sia online; se queste condizioni sono soddisfatte, la richiesta viene passata al pool sopra menzionato, che lancerà un thread per inoltrare la richiesta ad Y tramite datagtamma UDP.
Il thread in questione si sospende, rimanendo in attesa di una risposta per un

intervallo di *T1* secondi al massimo. In caso di risposta entro i *T1* secondi, comunicherà l'esito di accettazione o rifiuto all'utente *X*.

Durante l'attesa, il canale del richiedente viene settato momentaneamente con ***interestOps = 0*** nel corpo principale del server, in quanto rimane sospeso fino alla ricezione dell'esito. Quando il thread in attesa della risposta da parte di *Y* si sveglia, scrive sul canale di *X* l'esito e ripristina il suo readyOps ***interestOps(SelectionKey.OP READ)***.

2.2.2 Sfida

In caso di accettazione da parte dell'utente *Y*, viene inviato in automatico un messaggio di *start game* da parte del richiedente al server, il quale richiede l'esecuzione del game al pool. Parte così il setup della sfida. Subito prima di avviare la sfida, nel corpo principale del server, i due canali vengono momentaneamente sospesi settando i rispettivi ***interestOps = 0***.
Al thread adibito alla gestione della sfida, lanciato dal pool ***ThreadPoolExecutor games***, vengono passati i canali dei due sfidanti; a questo punto il suddetto thread apre un nuovo selettore sul quale registra i canali, ed avvia la sfida. Il match soddisfa precisamente quanto richiesto con qualche aggiunta:

1. Se i due utenti hanno totalizzato lo stesso punteggio ed hanno finito in tempo, i punti bonus vengono assegnati a chi termina per primo e di conseguenza vince la sfida.

2. Se i due utenti hanno lo stesso punteggio, ma non hanno finito in tempo, i punti bonus non vengono assegnati e di conseguenza non c'è un vincitore.

3. Se un utente chiudesse forzatamente il client durante una sfida, le successive parole non inviate dall'utente verrebbero considerate come sbagliate.

Al termine della sfida, dopo aver salvato e comunicato l'esito, questo selettore viene chiuso e vengono ripristinate le operazioni dei due canali col selettore principale del server.

2.3 Strutture dati e gestione concorrenza

Il server mantiene gli utenti registrati in ***ConcurrentHashMap<String, User> registeredUsers***, struttura condivisa con il task RMI di registrazione.
È stata presa la decisione di utilizzare questa struttura piuttosto che una classica ***HashMap*** perché, nonostante l'assenza di accessi concorrenti nel corpo principale del server implementato tramite NIO ed un selettore, ci potrebbero essere diversi accessi concorrenti in fase di registrazione RMI, con il rischio della probabile creazione di un thread per ogni client. Tuttavia, c'è la possibilità che si richieda un accesso concorrente a bucket diversi quindi, piuttosto che prendere la lock su tutta la struttura, viene presa sulla sezione desiderata, con un decisivo aumento delle performance.

Questa struttura è anche condivisa con il sopracitato pool adibito alle sfide. In fase di end game, diversi punteggi utente potrebbero essere in aggiornamento

contemporaneamente, e quindi la scelta di una ***ConcurrentHashMap*** elimina la necessità di bloccare tutta la struttura nonostante possa esserci bisogno di una sola sezione.

È stato eseguito un benchmark di prova che ha rafforzato la decisione di usare questa struttura:

- Registrazione contemporanea di 10000 utenti al servizio con annessa serializzazione su file.
    1. Utilizzo di ***ConcurrentHashMap<String, User> registeredUsers***, che sincronizza solo le sezioni desiderate impiegando 12 secondi.
    2. Utilizzo di una normale ***HashMap<String, User> registeredUsers*** che sincronizza esplicitamente tutta la struttura impiegando 84 secondi.

La directory **screenTest** contiene due file.png che mostrano le due diverse implementazioni ed i rispettivi risultati.

Il server mantiene gli utenti connessi in ***HashMap<String, UserItem> connectedUsers***; non essendoci accessi concorrenti a questa struttura grazie alla scelta implementativa NIO, non è mai necessario sincronizzarla.

Nel corpo principale del server una sola operazione richiede la sincronizzazione esplicita, l'operazione di ***mostra_classifica***. Più amici, dei quali X vuol conoscere i punteggi, potrebbero essere in fase di aggiornamento se si trovassero in end game, quindi, sincronizzando l'accesso ad essi, si ottiene la garanzia di avere sempre un valore consistente dei punteggi.
Di conseguenza, il thread che gestisce la sfida, prima di salvare i nuovi punteggi, sincronizza gli accessi agli sfidanti.

2.4 Classi

Il server, per implementare il servizio, fa uso di diverse classi; quest'ultime risiedono nel package **server** e verranno di seguito descritte brevemente. (Una descrizione più accurata dei vari parametri si trova nei rispettivi file.java)

1. **Server.java**: descritta inizialmente.
2. **RegisterImplementation.java**: realizza il task di registrazione degli utenti al servizio tramite RMI.
   Se i parametri passati sono corretti, crea un'istanza della classe **User** e di conseguenza lo inserisce in ***ConcurrentHashMap<String, User> registeredUsers***, in cui la chiave è l'username dell'utente, per poi salvare su file.
3. **User.java**: descrive l'utente ed i suoi attributi ed è identificato univocamente dal suo username.
4. **UserItem.java**: viene istanziata e messa come attachment alla ***SelectionKey*** dell'utente quando instaura una connessione TCP col server.

5. **ServerRequest.java**: task che inoltra le richieste di sfida da parte di un utente X ad un altro Y tramite UDP. Esso viene gestito da ***ThreadPoolExecutor requests***.

6. **Game.java**: task che rappresenta la sfida tra due utenti ed è implementato sfruttando **NIO**. Esso viene gestito da ***ThreadPoolExecutor games***.

7. **UserItemInGame.java**: estende **UserItem** aggiungendo alcuni parametri utili per la sfida. Viene istanziata e messa come attachment alla *SelectionKey* dell'utente durante la fase di setup del match.

8. **Dictionary.java**: dizionario per ogni sfida, che associa a K parole italiane le rispettive traduzioni ottenute dal sevizio esterno durante la fase di setup della sfida.

3 Client

3.1 Descrizione generale

La classe **Client.java** è stata definita come *abstract* in quanto, per i test forniti, il metodo ***game***, che gestisce la sfida tra due utenti, è leggermente diverso per ognuno di essi. I restanti metodi, quelli indicati nelle direttive del progetto, sono i medesimi per tutti i test, infatti questa classe implementa l'interfaccia **Operations** che contiene la definizione delle operazioni richieste.

Client si trova nel package **client** e presenta al suo interno le seguenti variabili di istanza:

```
1     // per connessione con server
2     private Socket socket;
3     // per input dal server
4     private BufferedReader in;
5     // verso il server
6     private DataOutputStream out;
7     // variabile di connessione
8     private boolean connected;
9     // thread che gestisce le richieste di sfida
10    private Thread requester;
11    // coda nella quale vengono inserite dal thread le richieste di
       sfida
12    private ArrayList<TimeOutDatagramPacket> requests;
13    // socket per inviare risposte di accettazione sfide
14    private DatagramSocket datagramSocket;
```

Listing 2: Client

Per richiedere una qualsiasi operazione al server che non sia *registrazione*, il client instaura una connessione TCP con esso attraverso ***Socket socket***.
Il messaggio viene inviato utilizzando ***DataOutputStream out***. La risposta viene letta attraverso ***BufferedReader in***.

3.2 Threads

Il client durante la sua esecuzione crea due thread in momenti distinti.

1. Una volta connesso al server ed eseguita correttamente l'operazione di *login*, crea il primo ***Thread requester*** adibito al salvataggio dei datagrammi UDP contenenti le richieste di sfida da parte degli amici in ***ArrayList<TimeOutDatagramPac*** ***requests***.

2. Il secondo viene creato subito prima di iniziare la sfida con un amico. Rimane in sleep per *T2* secondi, il tempo massimo di durata della sfida, e se allo scadere del timer non è ancora stato interrotto, comunica all'utente che non ha terminato in tempo.

3.3 Strutture dati e gestione concorrenza

L'unica struttura dati su cui soffermarsi è ***requests***. Questa lista è condivisa tra il ***main*** del client ed il ***Thread requester***, che rimane continuamente in ascolto sulla ***DatagramSocket datagramSocket*** in attesa di richieste di sfida. Quando la richiesta arriva, viene salvata nella sopracitata lista e, siccome condivisa, viene sincronizzata. L'utente, se desiderasse controllare le sue richiesta di sfida, interrogherebbe questa lista in maniera sincrona.

3.4 Classi

Il client, per implementare il servizio, fa uso di diverse classi localizzate nel package **client**, che verranno qui descritte brevemente. (Una descrizione più accurata dei vari parametri si trova nei rispettivi file.java)

1. **Client.java**: sopra descritta.
2. **TimeOutDatagramPacket.java**: contiene il datagramma di richiesta sfida, l'username del richiedente e l'intervallo di tempo di validità di esso.
3. **RequestListner.java**: viene istanziata ed eseguita da ***Thread requester*** quando il client esegue correttamente l'operazione di login; lo stesso thread rimane attivo per tutta la durata delle sessione, in attesa di ricevere richieste di sfida da amici dell'utente. Una volta ricevuto il datagramma UDP, crea un'istanza della classe sopra e lo inserisce nella coda condivisa **requests**.

4 Test

Sono state fornite tre diverse implementazioni che estendono la classe astratta **Client.java** per testare il servizio, implementando il metodo *game* in maniera leggermente differente, come descritto di seguito.

- In caso di test **ClientGUI.java**, le query del server vengono visualizzate in un *InputDialog* dove si attende anche la risposta dell'utente.
- In caso di **ManualTest.java**, le query del server vengono visualizzate a video e l'input è atteso da stdin.
- In caso di **AutoClient.java**, pensato esclusivamente per testare la solidità del servizio, le query non vengono visualizzate e le risposte sono inviate in maniera automatica.

4.1 Gestione eccezioni

Sono stati gestisti i casi di interruzione forzata nel seguente modo:

1. **lato server**: se un utente, una volta connesso, terminasse forzatamente l'esecuzione del client, questa interruzione verrebbe correttamente riconosciuta dal server che, conseguentemente, lo rimuoverebbe da ***registeredUsers***.
2. **lato client**: se il server terminasse improvvisamente, gli eventuali utenti connessi, alla successiva richiesta, verrebbero informati del crash del server.

4.2 ClientGUI

Questo test utilizza un'interfaccia grafica ed è composto da diversi ***JFrame***. L'interfaccia è minimale e guida lei stessa l'utilizzatore.
Il primo ***JFrame*** visualizzato all'apertura risiede nella classe **LoginGUI.java**. Presenta una classica schermata di login e l'opzione per registrarsi.

Una volta effettuato il login, viene istanziata la classe **ClientGUI.java** che rappresenta l'utente in sessione e viene aperto un nuovo ***JFrame*** che si trova in **ShellGUI.java**, dove sono visibili tutte le informazioni dell'utente nei diversi ***JPanel***.

Sia X l'utente attualmente connesso.

4.2.1 Threads

Verranno illustrati e spiegati di seguito i vari threads creati per implemenatre la GUI.

4.2.1.1 Visualizzazione ed accettazione richieste di sfida

Da notare il ***JPanel*** adibito alla visualizzazione delle richieste di sfida. Esso è aggiornato in tempo reale, ovvero, appena un utente Y amico di X, gli inoltra una richiesta di sfida, essa è immediatemente visibile.
Questo meccanismo è stato implementato nel seguente modo:
alla creazione di **ShellGUI.java**, dopo aver eseguito il login, viene avviato ***Thread visual*** che esegue il task **TaskUpdateRequestsJList.java**: per tutta la durata della sessione, ***visual*** rimane in attesa che in ***requests*** venga aggiunta una richiesta di sfida. Una volta svegliato, estrae il pacchetto, lo rimuove da ***requests***, e lo aggiunge alla ***JList<TimeOutDatagramPacket> matchRequestsJList*** in modo che sia subito visibile nella shell. Successivamente, sempre ***Thread visual***, richiede l'esecuzione di un altro task a ***ThreadPoolExecutor sleepAndRemove***: uno dei suoi thread deve attendere *T1* secondi e successivamente rimuovere la richiesta da ***matchRequestsJList*** per far sì che non sia più visibile se non è stata accettata.

Per quanto riguarda l'accettazione di una richiesta, se X decidesse di volerne accettare una, non dovrebbe fare altro che cliccare due volte sull'username dell'amico da sfidare. La richiesta verrebbe così rimossa dalla *JList* ed avviato un thread che gestisce la sfida.

4.2.1.2 Inoltro richieste di sfida

Quando X decide di sfidare un amico, inserisce il suo username nell'apposito *JTextBox* e clicca sul rispettivo *JButton*. Se la richiesta dovesse andare a buon fine, verrebbe visualizzato il *JFrame* istanziato dalla classe **WaitRequestGUI.java**. Questo *JFrame* mostra in tempo reale i secondi rimanenti di validità della richiesta.
Per implementare questa funzione si è utilizzato un thread che ogni secondo aggiorna il timer.

In questa classe si fa uso anche di un altro thread; prima si sospende finché non arriva l'esito della richiesta dal server e successivamente, in caso di esito positivo, richiama il metodo *game* della classe **ClientGUI.java**, altrimenti torna alla shell.

4.2.2 Concorrenza

Solo una struttura dati necessita di essere sincronizzata:
DefaultListModel<TimeOutDatagramPacket> requestsLM.
Alla stessa si potrebbe accedere in maniera concorrente in diverse occasioni:

1. Una richiesta di sfida arriva nel momento in cui X clicca due volte su un'altra richiesta per accettarla.

2. Una richiesta di sfida arriva nel momento in cui un thread del threadpool sopra descritto si è svegliato ed accede alla lista per rimuovere una richiesta scaduta.

3. La combinazione dei punti 1 e 2.

4.3 ManualClient

Questo test presenta un interfaccia testuale che ricalca l'esempio fornito nelle direttive del progetto. All'avvio viene mostrata la lista di operazioni disponibili e il client attende da input l'operazione desiderata. A questo punto l'utente può decidere se registrarsi o eseguire il login. La prima operazione per avere accesso al servizio dev'essere *login username password*; una volta ricevuto l'esito affermativo del server, comincia la vera e propria sessione.

Sia X l'utente appena connesso.
Se durante la sessione, un utente Y richiedesse di sfidare X, la richiesta non apparirebbe a video, ma verrebbe salvata nella sopracitata ***requests***. Se X desiderasse controllare le richieste, potrebbe eseguire l'operazione *mostra_ richieste*. Nel caso in cui dovessero esserci richieste, queste verrebbero mostrate a video con una rispettiva etichetta di validità e allo stesso modo verrebbe richiesta l'accettazione o il rifiuto di una di esse.

Nel caso in cui X voglia sfidare un altro utente, deve assicurarsi che sia online. Egli può controllare quale dei suoi amici sia online richiedendo al server l'operazione *amici_ online*. Una volta richiesta la sfida, X viene sospeso in attesa dell'esito che, se risulterà essere positivo, invierà al server un messaggio automatico che, a sua volta, farà partire la sfida fra i due utenti. Tutte le altre operazione sono quelle richieste e non necessitano di ulteriori descrizioni.

4.4 AutoClient

Questo test è stato pensato esclusivamente per effettuare uno stress test al server, per assicurarsi che il sistema sia stabile e per testare la concorrenza.
Esso simula in automatico diverse sessioni di utenti col server, ed è composto da tre fasi.

1. Si registrano 1024 utenti in contemporanea, lanciati da un apposito ***ThreadPoolExecutor***.

2. 64 degli utenti appena registrati eseguono il login ed attendono di essere sfidati. Le risposte alle query del server sono casuali per testarne la solidità. Al termine della sfida vengono richieste le altre operazioni e viene effettuata la disconnessione.

3. Altri 64 degli utenti registrati eseguono il login, aggiungono come amico un utente del punto 2, e lo sfidano. A termine sfida si disconnettono.

Nella directory **screenTest** si trovano due file.png che illustrano i risultati del benchmark illustrato al paragrafo **2.3**.
È presente anche un video che mostra una simulazione di sfida tra due client: **ClientGUI.java** vs **ManualClient.java**.

5 Istruzioni

Il progetto è stato svolto sull'IDE *IntelliJ IDEA*. La GUI è stata creata utilizzando i meccanismi che *intelliJ* mette a disposizione per le varie forme. Il progetto è stato successivamente convertito per essere utilizzato anche su *Eclipse* senza nessuna modifica.

All'interno della directory principale del progetto si trovano diverse componenti atte al funzionamento del servizio.

5.1 File testuali

1. **config.ini**: file di configurazione dal quale è possibile modificare i parametri della sfida ed il timeout delle richieste.
2. **users.json**: file *json* dove vengono salvati gli utenti registrati al servizio con le rispettive informazioni.
3. **words.txt**: file in cui si trovano un migliaio di parole italiane, K delle quali verranno estratte e tradotte per ogni sfida.

5.2 Librerie esterne

Nella directory **mylib** si trovano le due librerie esterne utilizzate:

1. **Gson.jar** con la rispettiva **JavaDoc.jar** per serializzare.
2. **forms_rt.jar** che permette di lanciare GUI create su *IntelliJ* anche su *Eclipse*.

5.3 Eseguibili

Tutti i packages e le classi si trovano nella directory **src**. Per lanciare il programma eseguire:

- **src**
  - **server**
    - **Server.java**

  Seguito da uno o più dei seguenti client:

  - **cli**
    - **AutoClient.java**
    - **ManualClient.java**
  - **gui**
    - **ClientGUI.java**