UNIVERSITÀ DI PISA

Relazione del progetto **WordQuizzle**

Nicolò Maio

RCL A.A. 2019-20

Indice

1 Scelte progettuali

Di seguito verranno illustrate le principali scelte progettuali effettuate durante la realizzazione del progetto.

1.1 Server

1.1.1 Architettura

L'architettura del server è stata scelta facendo un "mix" di due soluzioni, che sono state trattate più in dettaglio durante il Laboratorio di Reti di Calcolatori:

- multithread sincrona con I/O bloccande (Sockets di Java IO);
- monothread sincrona con I/O non bloccante (Selectors di Java NIO).

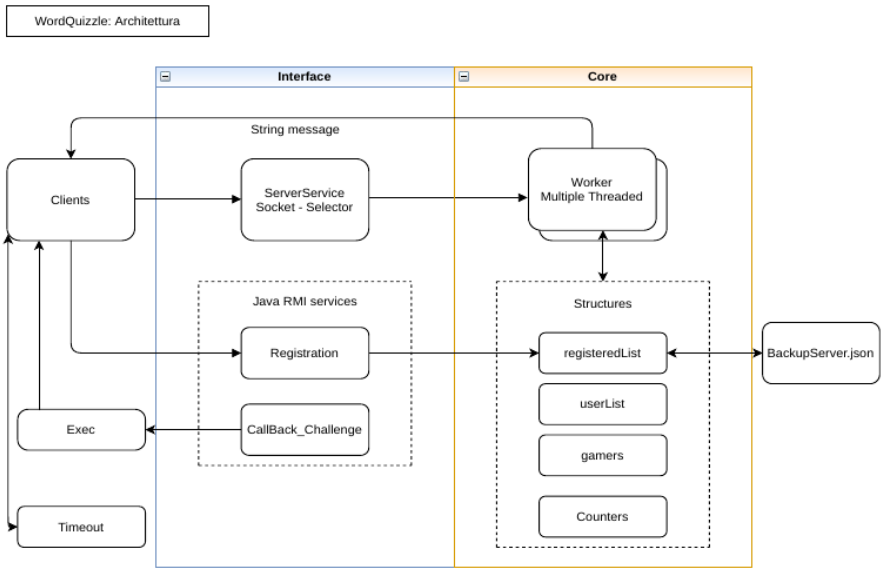
Combinando le due soluzione l'architettura di base risulta un selector di Java NIO che in caso di key con OP_READ registrata da ad un threadPoolExecutor le info sulla richiesta ottenuta da un client. Uno dei thread del pool si occuperà di gestire la richiesta.
Possiamo dividere il server in due livelli: **interfaccia** e **core**.

- A livello di interfaccia si trovano:
  - Il selector che gestisce le varie socket di connessione dei thread;
  - Il servizio RMI per la registrazione utente;
  - Il servizio RMI di notifiche push (Callback).
- A livello core si trovano:
  - Il thread pool che gestisce le richieste di ogni client;
  - Le varie funzioni per aggiornare le strutture dati relative agli utenti, e il file json di Backup chiamato BackupServer.json.

Per ogni client connesso il server non riserva un thread, ma per ogni richiesta di ogni client viene associato un thread del pool che si occuperà di gestire tale richiesta e spedire risposta al client.

1.1.2 Strutture dati

Le principali strutture dati utilizzate sono **alberi** (per gli utenti registrati con le loro info e per gli utenti online) e **tabella hash** (per i punteggi delle sfide). Verranno approfondite nella sezione di gestione della concorrenza.



1.1.3 Protocollo di terminazione

Quando il server viene terminato mediante inserimento da riga di comando della stringa "termina" o mediante SIGKILL, viene avviato il protocollo di terminazione che prevede:

- viene chiuso il socket del server;
- viene attesa la terminazione dei thread del pool per un tempo massimo pari a TimeUnit.MILLISECONDS.
- viene salvato il contenuto dell'albero contenente le info degli utenti nel file BackupServer.json, per mantenere uno stato da caricare al riavvio.

1.2 Client

1.2.1 Architettura

L'architettura del client è più semplice: effettua richieste in TCP sulla porta passata come argomento a momento d'esecuzione, all'host indicato come argomento e resta in attesa di risposta. Durante la normale esecuzione il client è single threaded; viene generato al bisogno un thread "Exec" nel caso in cui il client stia per ricevere una richiesta di sfida, dunque per avviare la comunicazione UDP; oppure viene generato un thread timeout per limitare il tempo per accettare la sfida o per limitare il tempo della durata della sfida una volta accettata. Infine un ulteriore thread "ShutDownerThread" viene invocato al momento della cattura dei segnali di terminazione.

Il client è stato sviluppato senza GUI ovvero come indicato nel pdf di assegnamento del progetto mediante le specifiche dei comandi da terminale.

1.3 Comunicazione client-server

Per la comunicazioen client-server è stata adottata una tecnica a scambio di messaggi sincrona. I messaggi sono in formato "operationType/username/otherInfo" e vengono spediti come stringhe.
Il formato dei messaggi è indicato nel file **message_format.txt**.

1.4 Librerie

Sia nel client che nel server è stata utilizzata la libreria json_simple per la creazione e il parsing di oggetti JSON.

2 Gestione della concorrenza

Il server da MainClassServer lancia un thread ServerService il quale resta in ascolto delle varie connessioni dei client che gestisce mediante selector e ThreadPoolExecutor. Ad ogni nuova connessione, il thread ServerService se la chiave è readable allora darà le info su questa ad un thread del pool che eseguira il task denominato **Worker** così che esso possa gestire la richiesta e la/le interazioni/e con il client.
Le strutture dati usate in mutua esclusione sono:

- TreeMap **registeredList**
- TreeMap **userList**
- Vector **gamers**
- HashMap contenuta in classe **Counters**

2.1 registerdList

Il server salva i dati dei vari utenti in una TreeMap chiamata **registeredList**, e ogni qualvolta essa viene aggiornata in mutua esclusione (usando blocchi syncronized) essa viene letta per riportare le proprie info in un file json di backup chiamato BackupServer.json. La registeredList ha come chiave gli username degli utenti e come valore un'istanza della Utente che conterrà le info dell'utente.

2.2 userList

Il server tiene traccia degli utenti online utilizzando una TreeMap chiamata **userList** avente chiave username dell'utente e valore SelectionKey dell'utente in questione. Essa viene modificata in mutua esclusione dai thread del pool mediante blocchi syncronized.

2.3 gamers

Il server memorizza inoltre in un Vector chiamato **gamers** gli username degli utenti che stanno svolgendo una sfida così che si possa impedire che un utente riceva notifiche per più di una sfida contemporaneamente.

2.4 HashMap contenuta in classe Counters

Infine il server salva i contatori delle sfide in una tabella hash istanziata all'interno della classe Counters, di cui parleremo nella sezione descrizione delle classi.

3 Descrizione delle classi

Di seguito una breve descrizione delle classi. Per approfondire, vedere il Javadoc alla pagina **index.html** nella cartella src/Documentation.

3.1 Classi condivise

- **ImplRemoteRegistration**: per la registrazione degli utenti;
- **NotifyEventImpl** e **ServerCallBImpl** per l'invio di notifica di sfida all'utente sfidato.

3.2 Classi del server

Il server oltre alle classi condivise è composto da:

- **MainClassServer**: la classe principale che lo inizializza (caricando le varie informazioni sugli utenti da BackupServer.json se esiste) e lancia il thread ServerService.
- **ServerService**: thread che gestisce le connessioni con i client mediante selector e threadPoolExecutor. Appena arriva una richiesta da un client dunque la chiave è readable esso lancia un Worker del pool con le varie info necessarie.
- **Worker**: implementa il task che può eseguire un threadPoolExecutor con le varie operazioni che il server deve garantire al client. (login, logout, sfida, etc...)
- **Utente**: contiene username, password, punteggio, lista amici (implementata con un Vector) di un utente.
- **Counters**: contiene una tabella hash con chiave: username e valore: int[]. L'array contenuto nel campo value della hashMap nominata table viene usato per memorizzare i contatori di una sfida.
  - int[0]: numero parole corrette.

- int[1]: numero parole sbagliate.
- int[2]: numero parole non tradotte.
- int[3]: punteggio totale.
- int[4]: indica se la sfida è stata completata o no.

3.3 Classi del client

Il client oltre alle classi condivise è composto da:

- **MainClassClient**: la classe principale che lo inizializza, instaura la connessione con server ed esegue interazione con interfaccia da linea di comando.
- **Timeout**: task di un thread timeout, al quale gli viene passato il numero di millisecondi da conteggiare mediante sleep. Viene usato per accettazione sfida e controllare durata sfida.
- **Exec**: task lanciato quando arriva notifica tramite callback nel caso in cui il client venga sfidato da un altro client.
- **ShutDownerThread**: task lanciato nel caso in cui venga intercettato un segnale di terminazione, si assicura che venga fatta prima della terminazione del client l'operazione di logout.

4 Testing

Il progetto è stato testato con successo con lla seguente configurazione: Manjaro su Intel Core i7-6500U.
Durante i test sono state riprodotte diverse condizioni, ad esempio:

- simulazione di più sfide contemporaneamente;
- arresto brutale di uno dei due client durante una sfida;
- arresto brutale del server;
- login di un secondo utente sotto stessa connessione;
- richiesta di sfida ad un utente che sta svolgendo un'altra sfida;
- vari test più semplici per testare le varie funzionalità offerte ad un client;

5 Manuale d'uso

Importante: per la compilazione è richiesto **Java 11** o superiore.

Compilazione Per compilare eseguite il comando dalla Cartella WordQuizzle:
javac
-sourcpath src/
-classpath lib/<nome della libreria>
src/*.java
-d ¡cartella di output¿

Esecuzione Sempre dalla cartella WordQuizzle, eseguire:
java
-classpath ¡cartella di output¿:lib/¡nome della libreria¿
MainClassServer [port]

Per il client sostituire MainClassServer con MainClassClient [host] [port].
Importante: Per separare le cartelle argomento di classpath, su Windows è necessario il punto e virgola, mentre su Linux i due punti.

5.1 Storia delle versioni

L'intera storia delle versioni è consultabile all'indirizzo: `github.com/NicoMaio/WordQuizzle`.