Relazione

Relazione sul progetto di Lab III **wordle**.

Struttura

Comunicazione client/server

Il programma è strutturato secondo il modelllo client/server, le classi di utility e le funzioni sono totalmente separati tra essi.

Il client ed il server comunicano tra loro attraverso messaggi, ho optato nel progetto nello scambio di messaggi mediante oggetti, in particolare i messaggi faranno tutti riferimento ad una classe `Messaggio` e per cui i diversi tipi di messaggi (richiesta di login, richiesta di share ecc.) saranno sottoclassi ed implementeranno attributi e metodi aggiuntivi a seconda del tipo di messaggio.

Schema

Ho cercato di dividere quanto più possibile i diversi compiti ed aspetti del programma in file diversi, per questo motivo: - Generalmente ogni classe che rappresenterà l'esecuzione di un thread per quanto piccola viene posta in un file diverso rispetto che inline. - Ogni tipo di messaggio scambiabile nella comunicazione è inserita in un file diverso. - Le classi Main di client e server sono "snellite" nella loro complessità creando una classe apposita che conterrà i modi in cui le due parti dovranno comunicare e le azioni che dovranno compiere, tutti i metodi di questa classe sono stati messi come metodi statici.

Configurazione

Sia client che server hanno la possibilità di parametrizzare e modificare vari funzionamenti del programma, per quanto riguarda il client la configurazione riguarda solo aspetti di connettività al server (indirizzi e porte) mentre per quanto riguarda il server vi sono anche aspetti prestazionali (numero di thread) e funzionalità (path per file di supporto).

Queste configurazioni sono racchiuse in due file `.properties` a cui client e server accederanno all'inizio della loro esecuzione per recuperare le opportune parametrizzazioni. Il programma si aspetta che questi file siano inseriti nella stessa cartella in cui avvine l'esecuzione del Client/Server.

Server

Strutture dati

Le due principali strutture dati utilizzate sono il database del login dei clienti ed il database delle partite giocate da essi.

Database utenti

Gli utenti vengono modellati in una classe `Utente` che contiene semplicemente nome utente, password e stato di login.

Il database è semplicemente realizzato mediante una lista di utenti nella classe `UsrDatabase`, con alcune operazioni di supporto.

Essendo una struttura dati condivisa da più thread essa possiede primitive di sincronizzazione che vengono discusse in una delle sezioni successive.

Database giochi

Le varie partite effettuate dagli utenti vengono salvate come stringhe all'interno della classe `Partita`, vengono anche salvati dati relativi alla parola segreta utilizzata ed altri attributi di supporto. Le varie partite sono raccolte in una struttura dati in una classe chiamata `DataGame` che associa ad ogni username una lista di partite, oltre a definire metodi di supporto utili per operazioni sulle varie partite.

I vari DataGame sono raccolti a loro volta in una classe chiamata `GameDatabase` la quale sarà la principale interfaccia con cui nel programma ci si approccia ai dati delle partite in quanto questa classe ha anche il compito di sincronizzare le varie operazioni di lettura/modifica delle partite. Gli aspetti di sincronizzazione vengono discussi più approfonditamente in una delle sezioni successive

Concorrenza

Client threads

Per i client ho optato l'utilizzo di ThreadPools in particolare la versione `FixedThreadPool`. Questo si adatta bene nell'utilizzo efficiente delle risorse per la gestione di vari client, con la possibilità della gestione contemporanea fino ad un massimo di n client contemporanei. Nel caso in cui vi siano più di n client in contemporanea il client $n+1$ dovrà aspettare e non sarà in grado di proseguire.

Avrei anche potuto usare un `CachedThreadPool` per poter riuscire a servire ancora più client oltre al limite inposto ma un `CachedThreadPool` potrebbe portare al dover gestire più client di quanti la macchina del server sarebbe effettivamente in grado di poter gestire.

Assumo che l'ordine di grandezza dei client possa essere stimato a priori e quindi ho optato per un `FixedThreadPool`.

Change thread

Viene creato un thread che leggerà dalla lista delle parole e le caricherà in memoria in una lista per poi restituire ad altri thread che lo richiedono la parola del momento. Nello specifico il thread adibito al chiedere le nuove parole da usare nel gioco è il thread `ChangeThread` che runna costantemente in background.

ExitHandler thread

Dato che prima che il server venga chiuso è opportuno salvare su file i cambiamenti effettuati a `gameDB` e `utentiDB` nel caso di un `SIGINT` si attiverà il `ShutdownHook` che creerà un nuovo thread che si occuperà dell'uscita pulita scrivendo in un file JSON i cambiamenti effettuati ai due database degli utenti e dei giochi.

Sincronizzazione

Database

I database contenenti i dati sugli utenti ed i dati dei giochi sono sincronizzati sugli oggetti che contengono i dati utilizzando il costrutto `syncronized(this)`. Vale a dire che non è possibile che due thread possa leggere/scrivere allo stesso tempo e che si presentino race conditions.

Wordlist

La lista delle parole è contenuta all'interno della classe `WordPicker` la quale verrà modificata periodicamente dal thread che esegue la classe `ChangeThread`. Per questo motivo vengono sincronizzati per evitare race conditions i metodi che settano e leggono la parola segreta. Non si ha bisongo però di sincronizzare il metodo che controlla se una parola è nella lista delle parole valide in quanto la lista è statica nel tempo.

Client

Strutture dati

Database notifiche

È presente per client una classe `NotificheDB` che rappresenta una coda di messaggi contenente le statistiche delle partite. Questa struttura dati si interfaccia con un thread che è sempre in ascolto per nuovi messaggi provenienti dal gruppo multicast nei modi descritti successivamente.

Concorrenza

McListener Thread

Questo thread runna perennemente la classe `McListenerThread`, la quale si unirà al gruppo multicast dato come parametro e starà perenemmente in ascolto per nuovi messaggi. Essa si occupa di ricevere i pacchetti UDP e poi castarli opportunamente come `StatisticMsg` opportuni per poi inserirli all'interno del database delle notifiche descritto sopra.

Sincronizzazione

Le uniche race conditions che possono accadere nel codice del client sono quelle riguardo all'aggiunta e rimozione di nuovi messaggi all'interno del database delle notifiche. Per questo motivo la classe `NotificheDB` è sincronizzata sull'oggetto stesso mediante il costrutto `syncronized(this)` in ogni metodo.

Compilazione ed esecuzione

Uso di script

Per facilitare il testing del programma sono presenti anche degli script in bash. Questi non vanno intesi come parte necessaria del programma ma solo come script utili per velocizzarne lo sviluppo. I più significativi di cui si riporta qui una breve spiegazione sono i seguenti:

runAll.sh

Questo script permette di far partire client e server su due nuovi terminali diversi chiamando a loro volta script che eseguono singolarmente il server ed il client. L'esecuzione tra server e client è intervallata da una sleep in maniera tale da assicurarsi che il server sia completamente in esecuzione prima che il client provi a connettersi.

jarAll.sh

Permette la creazione automatizzata dei jar di client e server in una cartella separata da quella principale di sviluppo.

Compilazione

Compilazione programma

Per facilitare e velocizzare il processo di compilazione ho creato dei makefile pertanto può essere sufficiente anche utilizzare il comando `make`. Nel caso si

voglia fare una compilazione manuale da linea di comando i comandi sono i seguenti:

Per quanto riguarda il server

```
# Assumendo di essere nella cartella con tutto il codice del server

javac -cp <path per gson-2.10.1.jar>:. *.java
```

Per quanto riguarda il client

```
# Assumendo di essere nella cartella con tutto il codice del client

javac *.java
```

Creazione jar

La creazione dei jar avviene a partire dai class file già creati con il comando precedente.

Per la creazione dei jar si ha bisogno di un file `manifest.txt` in cui bisogna scrivere alcune informazioni opportune. Di seguito questo file si mostra nella creazione interamente da linea di comando.

Per quanto riguarda il jar del server

```
# Si assume di essere nella cartella con tutti i file .class del server

echo 'Main-Class: WordleServerMain' > manifest.txt
echo 'Class-Path: ../gson-2.10.1.jar' >> manifest.txt
jar cvfm WordleServer.jar manifest.txt *.class <path per gson-2.10.1.jar>
```

Per quanto riguarda il jar del client

```
# Si assume di essere nella cartella con tutti i file .class del client

echo 'Main-Class: WordleClientMain' > manifest.txt
jar cvfm WordleClient.jar manifest.txt *.class
```

Esecuzione

Esecuzione normale

Per facilitare e velocizzare il processo di compilazione ho creato dei makefile con una clausola per runnare i programmi pertanto può essere sufficiente anche utilizzare il comando `make run`. In caso si voglia fare un'esecuzione manuale da linea di comando i comandi sono i seguenti:

Per quanto riguarda il server

```
# Si assume di essere nella cartella con tutti i file .class del server

java -cp <path per gson-2.10.1.jar>:. WordleServerMain
```

Per quanto riguarda il client

```
# Si assume di essere nella cartella con tutti i file .class del client

java WordleClientMain
```

Esecuzione da jar

Per quanto riguarda il server

```
# Si assume di essere nella cartella con il file WordleServer.jar
# e la libreria gson-2.10.1.jar nella posizione indicata da manifest.txt

java -jar ./WordleServer.jar
```

Per quanto riguarda il client

```
# Si assume di essere nella cartella con il file WordleClient.jar

java -jar ./WordleClient.jar
```

Dipendenze

Il progetto ha bisogno della libreria `gson-2.10.1.jar`.