RELAZIONE LABORATORIO DI RETI
A.A. 2019/20:
Word Quizzle di Renai Filippo 530478

LATO SERVER

Descrizione generale dell'architettura

Il server del progetto è stato realizzato in maniera multithreaded, quindi con l'utilizzo del package JAVA IO API, per quanto riguarda la trasmissione UDP ho utilizzato il datagram socket con ricezione bloccante.

Il server istanzia le strutture fondamentali per far funzionare la connessione, quali:

- reg → oggetto RMI utile per la registrazione.
- pool → oggetto ThreadPool che si occupa di gestire le richieste dei client.
- welcomeSocket → oggetto ServerSocket che si occupa di gestire le nuove connessione dei client.
- serverUdpSocket→ oggetto DatagramSocket che si occupa d'inoltrare le richieste di sfida via udp, per conto dell'utente e vedere la risposta.
- d→ oggetto DataBase che si occupa di gestire tutte le informazioni degli utenti e della loro persistenza.
- online → oggetto OnlineUtenti che si occupa di gestire tutte le informazioni degli utenti online e delle loro future sfide.

Ho scelto di utilizzare, per il server, il multithreaded per il semplice fatto che non occorre che il client non sia bloccante, eseguendo solo azioni sequenziali e non in parallelo, rendendo il progetto più facile anche nella lettura del codice, stesso ragionamento per l'UDP con ricezione bloccante. Per quanto riguarda il thread pool ho deciso di utilizzare newFixedThreadPool con 10 thread,dando quindi la possibilità di avere al massimo 10 utenti online nello stesso momento essendo un piccolo progetto.

Strutture dati e thread attivati

Tutte le classi che interessano le strutture dati sono raccolte nel package Dati.

Per la gestione di tutti gli utenti ho utilizzato una HashMap, dove la chiave è il nickname dell'utente, il nickname è unicovoco per la singola persona, il valore è una classe d'appoggio che contiene: la password,il punteggio e la lista degli amici gestita con una List di stringhe.

Per la gestione degli utenti online ho sempre utilizzato una HashMap,dove la chiave è il nickname dell'utente, il valore è una classe d'appoggio che contiene: la porta udp del socket del client, dove il server può inviare la richiesta di sfida,il punteggio di ogni singola partita, quando ci sarà una nuova sfida il punteggio viene azzerato, un booleano utilizzato per vedere se la sfida è terminata, e la lista di parole che l'utente deve tradurre in questa sfida.

Il file json che funge da database per gli utenti è così strutturato:
ex →{ "username" : ["password", punteggio,"amico1"….] }

La main class del server attiva il thread dedicato all'ascolto di richieste di connessione TCP da parte degli utenti, esso crea un task utilizzato per le varie richieste utente eseguito da un thread pool costituito da 10 thread.

Nel lato server non ci sono particolari problemi di concorrenza essendoci un task per ogni richiesta del utente, per quanto riguarda la sfida (non la richiesta) ho creato un metodo del client apposito dove indica lo sfidante, inizia la sfida del singolo utente, richiamo il booleano associato alla sfida dell'avversario per vedere se l'altro utente ha finito,dopodiché vedo chi ha vinto.

Descrizione classi

- MainClassWQServer: main class contenente metodo static.

main,il suo complito è inizializzare gli oggetti utili per il servizio.
- Listener: thread che si mette in attesa di connessioni da parte dei clients.
- RegistrationImpl: implementa l'interfaccia per la registrazione.
- RegistrationInterface: interfaccia per la registrazione degli utenti.
- UserRequestHandler: runnable che si occupa di gestire le richieste dei client.

Package Configurazione:
- ConfigServer: dove vengono memorizzate tutte le variabili utili per il server.

Package Dati:
- DataBase: dove sono memorizzati tutti gli utenti e si preoccupa della loro persistenza.
- UtenteApp: classe d’appoggio della classe DataBase dove sono segnate tutte le informazioni utili del singolo utente.
- OnlineUtente: gestisce gli utenti online.
- UtenteSfida: classe d’appoggio della classe OnlineUtente dove sono segnate tutte le informazioni utili per la sfida.

LATO CLIENT

Descrizione generale dell'architettura

Il client è stato realizzato attraverso l’utilizzo del package JAVA IO API. Per la trasmissione UDP ho utilizzato datagram socket con ricezione bloccante. Il client può accedere ai comandi offerti dal servizio mediante un’interfaccia a linea di comando.

Ho scelto di realizzare il client I/O bloccante, perché non ci interessa che sia non bloccante essendo tutte le operazioni sequenziali, stesso ragionamento per l’UDP bloccante e l’interfaccia a linea di comando perché non occorre particolarmente una interfaccia grafica per il tipo di servizio offerto.

Thread attivati

La main class del client attiva il thread,dopo il login di un utente, dedicato all'ascolto di richieste di sfida da parte degli altri utenti, tramite server, via UDP.

L'unico problema di concorrenza riguarda il fatto che non si puo' avere più scanner, ho risolto creando una variabile statica scanner nel main,di modo che tutti possano usufruirne, ed utilizzando una variabile locale occupato,settato nel thread listener che determina se lo scanner serve al thread listener che chiama il metodo gioco,oppure al main.

Ho utilizzato due booleani, uno statico inGame nella classe SupportClient, e inGame2 nella classe ListenerClient, utilizzate nel caso in cui un utente sta giocando e un altro utente vuole sfidarlo, semplicemente non risponderà e non apparirà su schermo la richiesta di sfida.

Descrizione classi

- MainClassWQClient: main class contentente il metodo static main, dove vengono prese le richieste da tastiera dell'utente che vengono gestite da un metodo statico esterno.
- SupportClient: prende le richieste di operazioni consentite dalla classe principale, comunica con il server e stampa la risposta.
- ConfigClient: dove vengono memorizzate tutte le variabili utili per il client.
- RegistrationInterface: interfaccia per la registrazione degli utenti.
- ListenerClient: thread attivato al momento del login di un utente, che si mette in attesa d'un invito da parte dei server di sfida via UDP.

Compilare ed eseguire entrambi i lati

Nella cartella altro troviamo:

- la libreria esterna jar JSON Simple, ho utilizzato questa libreria per quanto riguarda la parte che utilizza json.
- Il dizionario formato txt contente le parole italiane da tradurre per la sfida.
- il database formato json contente già qualche utente, utili per testare il progetto.

Per testare il progetto ho utilizzato interamente eclipse sia lato server,che lato client.

I valori espressi come K, N, T1, T2, X, Y e Z sono stati settati nei file configurazione lato client e lato server.

L'interfaccia a linea di comando che ho utlizzato è la stessa nella descrizione del progetto:

```
$ wq --help
usage : COMMAND [ ARGS ...]
Commands:
registra_utente <nickUtente > <password > registra l' utente
login <nickUtente > <password > effettua il login
logout effettua il logout
aggiungi_amico <nickAmico> crea relazione di amicizia con nickAmico
lista_amici mostra la lista dei propri amici
sfida <nickAmico > richiesta di una sfida a nickAmico
mostra_punteggio mostra il punteggio dell'utente
mostra_classifica mostra una classifica degli amici dell'utente (incluso l'utente stesso)
```

I file database.json e Dizionario.txt l'ho posizionati nel Desktop,quindi si dovrà cambiare il path di entrambi i file (modificarli nei file di configurazione sia lato server che client).

Per testare il progetto basta compilare ed eseguire le due classi principali di entrambi i lati, MainClassWQServer e

MainClassWQClient aggiungendo la libreria esterna JSON Simple che ho chiamato json.jar .

Server

```
$ javac -cp ".;json.jar" MainClassWQServer.java
$ java -cp ".;json.jar" MainClassWQServer
```

Client

```
$ javac -cp ".;json.jar" MainClassWQClient.java
$ java -cp ".;json.jar" MainClassWQClient
```

Ovviamente il jar deve essere nella stessa directory.