RELAZIONE PROGETTO - LABORATORIO 3

STUDENTE: Federico Gueli **MATRICOLA**: 672392

1.Descrizione generale delle strutture, dei thread e delle scelte seguite lato SERVER

Il server è stato progettato seguendo un'architettura multithreaded orientata agli eventi. Per garantire ordine e anche una maggiore manutenibilità del codice, ho diviso il sistema in "Gestori", ognuno dei quali è responsabile di una specifica area funzionale.

1.1 GestoreServer

Questa classe è il centro dell'organizzazione, le sue responsabilità sono:

- Gestione della connessione: L'apertura del canale di comunicazione, la socket, tra il server e il client. Le comunicazioni vengono gestite come viste a lezione, tramite un newCachedThreadPool così da creare un thread per ogni nuovo utente che si collega al gioco, al quale passo il runnable GestoreClient.
- Gestione delle partite: viene creato un runnable LettoreJson che viene passato ad un thread che periodicamente legge le partite dal file e carica le parole in una mappa con chiavi le parole, e per valori i temi.
- Gestione dei client connessi: ogni client collegato viene aggiunto ad un CopyOnWriteArrayList, utile anche per il contesto multihreaded in cui ci troviamo

1.2 GestoreUtenti

- Struttura dati: all'interno del gestore è stata utilizzata una concurrentHashMap, per garantire un accesso rapido ai dati di ogni utente tramite lo username (che nel nostro contesto funge da chiave per la mappa), ma anche per garantire sicurezza ai dati salvati
- Persistenza: i dati che vengono salvati sul file. Il salvataggio all'interno del file avviene però in momenti specifici, e non per ogni mossa svolta dall'utente. Ad ogni mossa infatti salviamo solo i dati all'interno della map, per avere consistenza dei dati, ma scriviamo su file solamente quando un utente vince o perde la partita, o quando effettua un'operazione di logout.
- Stato Utente: La classe Utente memorizza non solo le credenziali e i dati globali come il punteggio, ma anche le statistiche temporanee che riflettono lo stato dell'utente nella partita corrente. Questi dati sono fondamentali anche per ripristinare il corretto stato di un utente in caso di crash del sistema, ma anche più semplicemente quando un utente decide di loggare nella stessa partita nella quale aveva effettuato un logout.

1.3 GestoreStorico

- Similmente a quanto detto per il gestore utente, questa classe si occupa di mantenere l'archivio di tutte le partite passate, salvandole nel file storico_partite.json. Permette anche il recupero delle statistiche e delle informazioni utili che l'utente può richiedere.

1.4 Lettore Json

- All'interno di questa classe viene gestita la lettura del file contenente le parole da utilizzare in un dato momento tramite una strategia basata sulle Stream API (Gson). E' stata utilizzata questa tecnica per l'efficienza che la contraddistingue, infatti possiamo evitare di carica in memoria tutto il file e leggere sequenzialmente le parti che ci interessano. La struttura risultante ad ogni iterazione è una mappa, contenente le parole della partita da giocare con i rispettivi temi come valori. Ho seguito questo stile così da poter accedere subito (in O(1)), dopo che un client ha mandato una proposta di soluzione, alle parole corrette salvate nella mappa, per rendermi conto non solo se la proposta è errata, quindi sono state raggruppate parole non appartenenti allo stesso gruppo, ma anche se sono stati presenti errori grammaticali nella proposta, caso che deve essere gestito diversamente.

1.5 GestoreClient

La classe implementa l'interfaccia Runnable e gestisce l'interazione diretta tra il singolo Client connesso e il server:

- Si occupa del parsing dei messaggi mandati dal client, così da poter permettere al server di rispondere correttamente alle operazioni richieste dall'utente

2.Descrizione generale delle strutture, dei thread e delle scelte seguite lato CLIENT

2.1 ClientMain

- Legge dai file di configurazione il numero di porta e l'indirizzo ip del server per un corretto collegamento con quest'ultimo.
- Utilizza un loop (while(true)) per gestire i comandi inviati da linea di comando dall'utente.
- I messaggi così impacchettati correttamente per il tipo di operazione da effettuare, vengono aggiunti da una coda bloccante (BlokingQueue) per poi essere successivamente mandati dal thread che si occupa del collegamento di rete

2.2 Nio Client

Gestisce la comunicazione con il server utilizzando un Selector e una SocketChannel in modalità non bloccante

- Scelta tecnica per la scrittura: Ho scelto di implementare la scrittura sulla socket direttamente accedendo a quest'ultima, senza l'uso di OP_WRITE per una serie di motivi: l'approccio diretto risulta infatti più semplice e anche leggermente più efficiente, dal momento che, essendo i messaggi inviati di piccoli dimensioni, il buffer di invio si riempirà con bassissima probabilità, portando il selector a stare indefinitamente in OP_WRITE se non ci fosse un controllo che disattivi subito OP_WRITE dopo aver scritto il messaggio.

2.3 GestoreProtocollo

Classe che si occupa della visualizzazione del messaggio json mandato dal server sulla console dell'utente

3. Altre classi utili per lo scambio di informazioni tra client e server

La comunicazione avviene tramite lo scambio di messaggi json. Per facilitare la serializzazione e la deserializzazione (sempre tramite la libreria gson), ho creato delle classi apposite per gestire ogni operazione richiesta dall'utente:

- RichiestaUtente: per login, registrazione e logou
- PropostaSoluzione: per l'invio delle parole raggruppate dall'utente
- RichiestaPartita: per la richiesta delle info e delle statistiche della partita corrente o di una partita nello specifico
- RichiestaClassifica: per la richiesta della classifica globale, della classifica con i primi k utenti o per la richiesta su un singolo utente
- AggiornamentoCredenziali: per la richiesta di aggiornamento delle proprie credenziali
- RispostaJson: formato standard con il quale il server invia il messaggio di risposta e le parole della partita

4. Sintassi dei comandi da poter utilizzare durante una partita

- submitProposal p1,p2,p3,p4
- requestGameInfo [id] (se un utente vuole le info su una partita nello specifico allora si deve specificare l'id)
- requestGameStats [id] (stessa cosa per il caso di info)
- login <username> <password>
- register <username> <password>
- updateCredentials <vecchioNome> <vecchiaPass> <nuovoNome> <nuovaPass>
- requestLeaderBoard (senza specificare altro verrà mostrato la classifica globale con tutti gli utenti)
- requestLeaderBoard k (specificando un intero si mostrano invece i top k utenti)
- requestLeaderBoard <username> (si specifica invece il nome utente di qui si vuole conoscere le statistiche)
- logout
- requestPlayerStats

5. Istruzioni per la compilazione ed esecuzione del programma

All'interno della cartella lib è presente il file .jar per la libreria Gson

5.1 Il comando da eseguire per la compilazione è:

- Nei sistemi Linux: javac -d bin -cp ".:lib/*" server/*.java client/*.java
- Nei sistemi windows: javac -d bin -cp ".;lib/*" server/*.java client/*.java

5.2 Per creare gli eseguibili JAR, utilizzare i file manifest forniti, nei quali sono specificati anche i classPath per rendere poi l'esecuzione più semplice

- Server: jar cmf manifest.txt server.jar -C bin server
- Client: jar cmf manifestClient.txt client.jar -C bin client

5.3 Infine per eseguire e avviare l’applicazione:

- Server: java -jar server.jar
- Client: java -jar client.jar