Documentazione Progetto Laboratorio III

New York Times Connections Clone

Biagio Montelisciani 676722

2026-01-08

Contents

1 Introduzione

A seguire il mio progetto Java per l'esame di Laboratorio III. Nel progetto implemento l'architettura client-server multithread con Java NIO per implementare un clone del gioco Connections di New York Times.

2 Architettura del sistema

Come da requisiti, il sistema è client-server e supporta molteplici client connessi allo stesso server. Ogni client è in grado di connettersi, registrarsi, fare login e tutte le altre operazioni attraverso comandi scritti su linea di comando. Approfondirò i comandi in seguito. L'unica libreria esterna utilizzata è GSON di google, vista a lezione.

2.1 Client

i pacchetti del client sono `ui`, `network` e `logic`; `ClientMain` avvia i vari thread e `ClientConfig` fa il parsing del file di configurazione `client.properties`.

`ui` gestisce l'interazione con l'utente tramite linea di comando. Ho implementato una semplice Text User Interface con alcuni colori ANSI per distinguere informazioni più velocemente e per estetica. All'avvio si vede subito una tabella che contiene tutti i comandi disponibili. Ogni comando ha un alias breve per essere scritto più velocemente. `/help` o `/h` fa stampare la **lista completa di comandi** e cosa fanno. **Per inviare le parole durante una partita basta scriverle a riga di comando**. Se una parola contiene spazi si può scrivere tra virgolette. Se il comando `/register` di un nuovo utente va a buon fine, il login è automatico.

Ci sono due comandi nascosti, impossibili da conoscere senza leggere il codice, che richiedono una password per essere chiamati. La password è `123` e si può modificare in `server.properties`.

`/oracle <pwd>` rivela le parole vincenti della partita attiva

`/god <pwd>` stampa la lista degli utenti registrati e le loro password

`network` si occupa della comunicazione con il server, usa Java NIO e di fatto invia richieste o riceve risposte. Ogni risposta ricevuta è mostrata attraverso la TUI e ogni richiesta inviata è inviata dal `CommandProcessor` (vedi prossimo paragrafo). Ogni risposta è di tipo `ServerResponse` e contiene un codice che ne identifica il sottotipo.

`logic` contiene la classe `CommandProcessor`, che si occupa di elaborare i comandi provenienti dalla `ui`. Ogni comando è contenuto in una mappa `<stringa, handler>` sfruttando un'interfaccia funzionale. In questo modo i comandi possono essere dichiarati tutti insieme e gli si possono assegnare informazioni da stampare nel tutorial con `/help`. Tutte le richieste client che sono fatte nel

pdf hanno uno o più comandi dedicati, ad esempio `/info` manda informazioni sulla partita attuale e `/gameinfo <id>` le manda su una vecchia partita, mentre `/gamestats <id>` manda le statistiche di una partita.

2.2 Server

Il server gestisce la comunicazione con molteplici client, la logica di gioco centralizzata e la sincronizzazione dello stato. L'architettura è multithreaded e si basa su `java.nio` per gestire le connessioni TCP in modalità non bloccante tramite un `Selector`.

I pacchetti del server sono `network`, `handlers`, `services` e `models`; il punto di ingresso è `ServerMain`, che si occupa di caricare la configurazione da `server.properties`, inizializzare i servizi di persistenza e avviare il thread di rete e lo scheduler di gioco.

`network` gestisce la comunicazione. La classe `NetworkService` inizializza il `ServerSocketChannel` e il `Selector`. I messaggi in arrivo vengono letti, assemblati (gestendo eventuali frammentazioni TCP con `PacketHandler`) e deserializzati da JSON in oggetti `ClientRequest`. Una volta ricostruita la richiesta, questa verrà elaborata, ma non dalle risorse in `network`. Questo pacchetto gestisce anche l'invio delle risposte (`TcpWriter`) e le notifiche asincrone via UDP (`UdpSender`).

`handlers` contiene la logica vera e propria. Riceve una richiesta specifica (es. `Login`, `SubmitProposal`, `RequestGameStats`) e interagisce con i servizi per produrre una `ServerResponse`. Ho separato le responsabilità in classi diverse: `AuthHandler` gestisce registrazione e login, `GameHandler` valuta i tentativi di gioco e la vittoria/sconfitta, `StatsHandler` calcola le statistiche e `InfoHandler` fornisce lo stato delle partite.

`services` ospita i gestori dello stato condiviso, implementati come Singleton thread-safe. `GameManager`: Mantiene il riferimento al `GameMatch` corrente e allo storico delle partite giocate (`matchHistory`). Collabora con il `GameScheduler`, un thread dedicato che monitora il tempo trascorso e caricale nuove partite dal file JSON quando il tempo scade, garantendo che tutti i client giochino la stessa partita nello stesso momento. `UserManager`: Gestisce l'autenticazione e l'aggiornamento dei profili utente in memoria.

`models` definisce le entità del dominio. `Game` è una partita, come quelle contenute in `Connections_Data.json` `GameMatch`: Rappresenta l'istanza di una partita in corso. Contiene un riferimento a tutti i giocatori attivi in quella specifica sessione. `PlayerGameState`: Mantiene lo stato individuale di un giocatore per una specifica partita (parole già indovinate, errori commessi, flag di vittoria).

La gestione della concorrenza è trasversale a tutti i pacchetti. ho usato`ConcurrentHashMap` nei manager e sincronizzazione nei metodi critici di modifica dello stato (es. aggiornamento punteggi o registrazione utenti).

Il server ha una UI minima, per uscire è sufficiente digitare `exit`.

2.3 Persistenza e Accesso a Dati

Come da richieste, il file `Connections_Data.json` è trattato come un file di grandi dimensioni e quindi vi si accede tramite stream senza caricarlo tutto in memoria. I `Game` vengono acceduti in ordine. e processati, poi Immagazzinati in memoria e periodicamente scritti in un file JSON `GameHistory.json`. Ai fini di questo progetto accademico ho trovato adatta la scelta di visualizzare a stream i game nuovi ma mantenere in memoria con una `ConcurrentHashMap` quelli vecchi per essere acceduti quasi istantaneamente e senza la ricerca lineare imposta dallo stream.

Simile strategia è adottata per gli utenti: in `Users.json` sono contenuti i dati di ogni utente aggiornati periodicamente e caricati all'avvio del server in caso di caricamento di vecchi user. Anche essi sono contenuti in una `ConcurrentHashMap` ed hanno un id univoco oltre al loro `username`, questo per non rendere un valore modificabile chiave della mappa.

3 Compilazione & Esecuzione

Il progetto è fornito completo di `Makefile` per l'automazione del build. Tuttavia, è possibile eseguire direttamente gli artefatti già prodotti o ricompilare manualmente il codice sorgente utilizzando gli strumenti standard del JDK (`javac`, `jar`).

3.1 Esecuzione Rapida

Nella cartella `/bin` sono già presenti i file eseguibili (`server.jar` e `client.jar`) compilati e pronti all'uso. Li ho fatti con il makefile e sono già legati alla libreria GSON.

```
java -jar bin/server.jar
java -jar bin/client.jar
```

In terminali diversi. Poi si puossono aprire anche altri client.

3.2 Compilazione ed Esecuzione manuali

Per compilare con `javac`:

```
mkdir -p bin
javac -d bin -cp lib/gson-2.11.0.jar $(find src -name "*.java")
```

Per creare i `.jar` non connessi a GSON:

```
# Creazione server.jar
jar cfe server.jar server.ServerMain -C bin .
```

```
# Creazione client.jar
jar cfe client.jar client.ClientMain -C bin .
```

Per eseguire (Linux/Mac)

```
java -cp server.jar:lib/gson-2.11.0.jar server.ServerMain
java -cp client.jar:lib/gson-2.11.0.jar client.ClientMain
```

Per eseguire (Windows) non ho potuto testare perchè non possiedo un computer Windows.

```
java -cp "server.jar;lib/gson-2.11.0.jar" server.ServerMain
java -cp "client.jar;lib/gson-2.11.0.jar" client.ClientMain
```