Relazione Progetto Reti 19/20

Alessio Loddo 560008
Corso A

Introduzione

Implementazione del progetto finale del corso “Word Quizzle”. Word Quizzle è un gioco di traduzione di parole con un lato social. Ogni utente può aggiungere altri utenti alla sua cerchia di amici e sfidarsi tra di loro.

Struttura

Il progetto è diviso in due moduli: client e server. Il modulo client mette a disposizione le funzioni di Word Quizzle all’utente attraverso un’interfaccia grafica semplice da usare; il modulo server gestisce i vari client e li coordina durante le sfide.

Server

Il server è implementato come applicazione su linea di commando e non necessita di alcun input. Le porte di ascolto sono predefinite: 8080 per la connessione TCP e 8888 per il registry RMI. UDP non ha una porta predefinita in quanto non viene usato per la ricezione.

Struttura

Il server è strutturato a strati: lo strato più basso è composto dalle classi User, UserManager, Sfida, SfidaManager, WordManager, Stats, UserInstanceData, ScoreboardRecord e Utils. Questo strato si occupa di mettere a disposizione le funzionalità essenziali al livello superiore.

Le classi Sfida, User, ScoreboardRecord, UserInstanceData e Stats sono delle semplici strutture dati per salvare le informazione relative a sfida, utente, record di classifica, istanza utente e statistiche rispettivamente. In particolare UserInstanceData viene usata dallo UserManager per gestire dati di istanza di un utente loggato come per esempio la sua socket.

La classe Utils è stata inizialmente pensata per contenere funzioni varie di utilità. Correntemente contiene solo la funzione di traduzione delle parole da italiano a inglese attraverso il web service specificato nella consegna.

La classe UserManager si occupa della gestione degli utenti; essa tiene traccia degli utenti registrati con le loro relative informazioni, gli utenti loggati in uno specifico istante e si occupa della permanenza dei dati salvandoli su un file data.json contenuto nella cartella data. Mette inoltre a disposizione delle funzioni di utilità come per esempio getClassifica(String username) che restituisce la classifica composta dalla cerchia di amici dell’utente username e isUserRegistered(String username), usata dal server durante la fase di registrazione di un utente, che controlla se un utente è già registrato.

La classe WordManager gestisce il dizionario delle parole. Essa carica le parole italiane dal file parole.txt contenuto nella cartella data all’avvio e le conserva in memoria per accesso veloce. Mette inoltre a disposizione le utility per la traduzione delle parole e il retrieve di una lista casuale di parole. Viene usato in fase di creazione di una sfida.

La classe SfidaManager gestisce lo storage e la creazione delle sfide. In questo caso per sfida si intende solo i dati relativi ad essa, ovvero la lista di parole da tradurre, la lista di parole tradotte, gli oggetti di sincronizzazione e il nome dell’utente che ha generato la sfida che viene usato come id della sfida stessa.

Lo strato superiore è composto dalle classi MainClass, Server, ClientThread e dall'interfaccia RegistrationInterface.

La classe MainClass è la funzione che contiene l'entry function dell'applicazione. Il suo unico scopo è inizializzare il server.

L'interfaccia RegistrationInterface mette a disposizione il metodo di registrazione per RMI.

La classe Server mette a disposizione le funzioni principali del server, ovvero quelle specificate nella consegna come lista_amici(nickUtente), mostra_punteggio(nickUtente) etc.., e si occupa di rimanere in ascolto per connessioni da nuovi client. Ad ogni client connesso viene assegnato un thread di tipo ClientThread.

La classe ClientThread implementa il thread per la gestione dei client e mette a disposizione le funzioni del Server ad esso. Le interazioni avvengono attraverso un semplice protocollo testuale brevemente descritto in seguito.

Scelte di implementazione

Protocollo di comunicazione

Il protocollo prevede come primo messaggio una stringa rappresentante un commando terminata da \n, es. "login\n", "aggiungi_amico\n", etc.., a seconda del commando seguono stringhe successive contenenti i parametri. Per esempio dopo "login\n" seguono "username\n" e "password\n". Questo protocollo è stato progettato per essere il più semplice possibile e facilmente leggibile da un umano, in modo da semplificare il debugging e la lettura del codice.

Fase di sfida

Quando un client C1 richiede una sfida ad un utente U2, il suo thread entra in modalità sfida. A questo punto il thread T1 genera una nuova sfida (contenente tra le altre cose gli oggetti di sincronizzazione), cerca i dati di istanza di U2 (dopo i dovuti controlli), preleva la porta di ascolto UDP di C2, gli invia la richiesta di sfida e si mette in attesa sull'oggetto waitForFriend della sfida appena creata con un timeout di 10 secondi; allo scadere del timeout la richiesta viene considerata rifiutata.

Il client C2 riceve la richiesta e a questo punto può decidere se accettarla o rifiutarla inviando al server rispettivamente "accetta_sfida\n" o "rifiuta_sfida\n" seguita dal nome utente di chi gli ha inviato la richiesta. Il thread T2 riceve il commando, prende la sfida generata da T1 attraverso SfidaManager, imposta il campo accepted a true o false e notifica T1 attraverso l'oggetto waitForFriend;

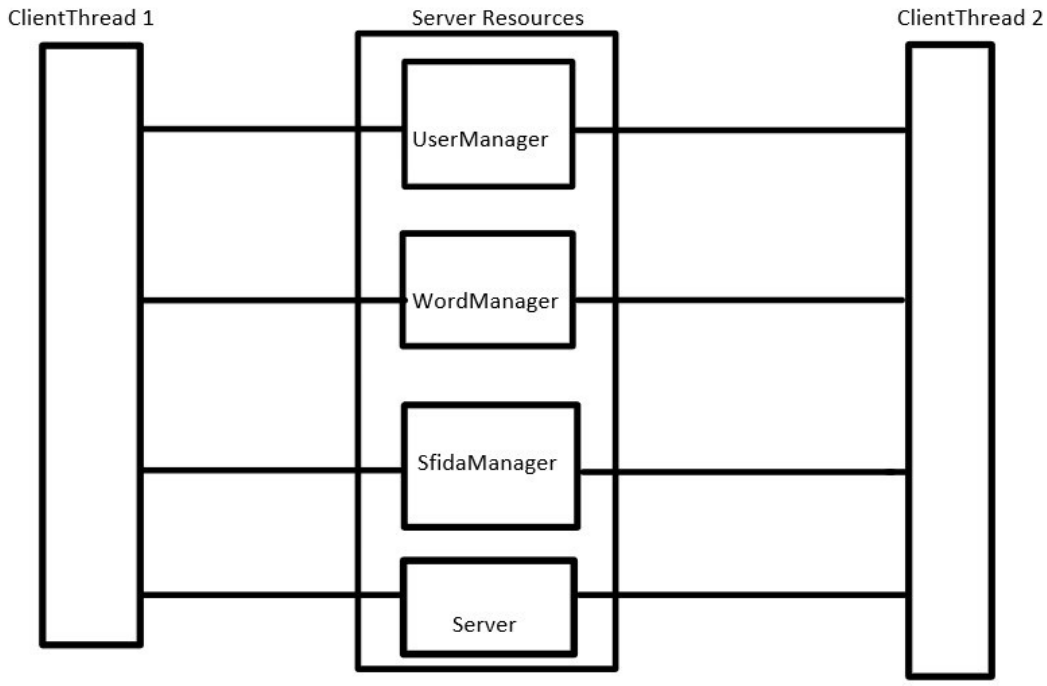
Se C2 ha accettato allora entrambi i thread passano in fase di sfida e vengono gestiti indipendentemente. Una volta che la sfida è finita (per timeout o parole finite) i thread controllano se l'altro ha finito attraverso il campo onePlayerDone di sfida. Se l'avversario deve ancora finire allora il thread si mette in wait sul campo endWait della sfida, altrimenti notifica l'altro attraverso lo stesso oggetto. A questo punto il server comunica gli esiti ai client.

Per gestire il timeout i thread in fase di sfida hanno il socket timeout impostato a 1 secondo; ogni timeout controllano quanto tempo è passato. Se sono passati più di 30 secondi dall'inizio della sfida allora mandano un messaggio di timeout ai client e terminano la sfida.

Calcolo del punteggio

Vengono assegnati +2 punti ad ogni risposta esatta, -1 punto ad ogni risposta sbagliata, 0 ad ogni risposta non data e +5 punti al vincitore.

Schema dei thread



I thread interagiscono con le risorse del server attraverso i manager, i quali garantiscono che non ci siano problemi di concorrenza attraverso i metodi syncronized. E` possibile modificare una risorsa senza passare dai manager una volta prelevata ma questo non pone un problema, in quanto queste risorse vengono gestite da un solo thread. Per esempio un ClientThread preleva il suo utente attraverso lo UserManager e né modifica un campo. Questo non pone problema in quanto ogni thread ha uno user diverso.

Client

Il client è implementato con un'interfaccia grafica; appena aperto richiede l'indirizzo IP per la connessione al server (la porta del server è fissa). Se la connessione ha successo allora si apre la schermata di login/registrazione . Dopo la registrazione si apre la schermata principale; a sinistra è presente la sidebar con i tasti di navigazione per le varie schede, mentre a destra c'è la scheda aperta al momento (di base la scheda amici). Uno schema più generale dell'interfaccia viene dato in seguito. Il client usa porte casuali, eccetto per il socket UDP; quel socket usa come porta il numero di porta locale TCP + 1.

Struttura

Le classi del client possono essere divise in due gruppi: un gruppo relativo all'interfaccia e un gruppo relativo alle funzionalità del client

Nel secondo gruppo ci sono le classi Client, MainClass, ChallengeThread, ScoreboardRecord e l'interfaccia RegistrationInterface.

La classe Client contiene tutte le funzionalità del client implementate attraverso metodi come login, aggiungiAmico, etc…

La classe ChallengeThread implementa il thread il cui scopo è rimanere in ascolto per eventuali richieste di sfida. Dispone inoltre di uno storage per le richieste in sospeso.

La classe MainClass contiene l'entrypoint dell'applicazione. Si occupa solo di inizializzare la finestra principale.

La classe ScoreboardRecord viene usata per gestire i record della classifica.

L'interfaccia RegistrationInterface mette a disposizione il metodo di registrazione per RMI.

Per le classi del primo gruppo segue una breve descrizione:

MainWindow è la finestra principale che funziona come contenitore per le altre. Si occupa anche di inizializzare il client.

Colors contiene dei colori standard per l'interfaccia.

ConnectWindow è la prima finestra e si occupa della connessione con il server.

LoginWindow è la seconda finestra e si occupa del login e della registrazione.

FriendWindow è una delle tre finestre centrali, mostra la lista amici e si occupa di inviare richieste di sfida e di aggiungere nuovi amici.

ScoreboardWindow è un'altra delle finestre centrali, mostra la classifica del circolo degli amici.

RequestWindow è l'ultima delle finestre centrali, mostra le richieste in sospeso e si occupa di accettarle o rifiutarle.

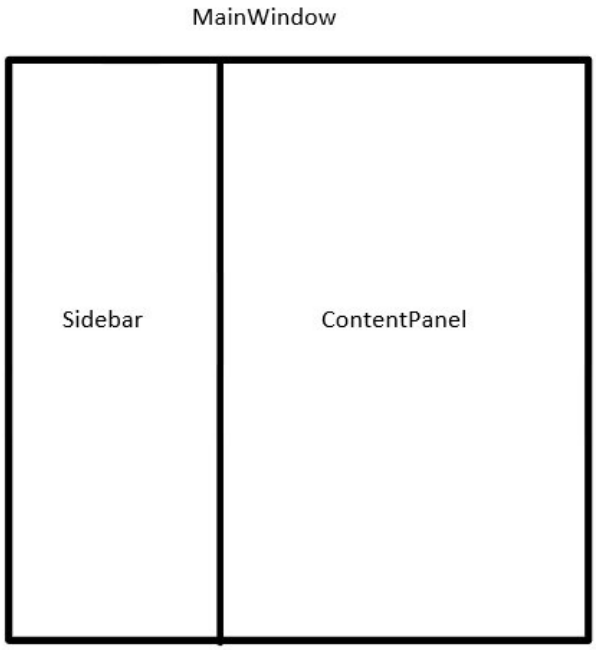
ChallengeWindow è la finestra che gestisce la fase di sfida.

SidePanel è l'implementazione della sidebar e si occupa della navigazione tra le finestre centrali.

Le classi SimpleButton, SimpleLabel e WideButton sono delle versioni stilizzate delle loro controparti

EventListener è un'interfaccia per la gestione degli eventi tra finestre.

UI



Schema generale

La classe MainWindow gestisce la finestra e funziona come un contenitore per le altre finestre. Le altre classi Window vengono viste come dei moduli da inserire all'interno del ContentPanel, mentre la sidebar ha

lo scopo di finestra di navigazione per selezionare i diversi moduli (per connessione, login e sfida la sidebar è disattivata e ContentPanel prende tutta la MainWindow).

Ogni modulo gestisce la propria business logic. Per esempio il modulo FriendWindow si occupa della gestione degli amici senza mai usare funzioni degli altri moduli, neanche di MainWindow.

I vari moduli interagiscono con la MainWindow attraverso un sistema ad eventi. Ogni modulo ha un oggetto di tipo EventListener che usa nel momento che vuole comunicare qualcosa. Per esempio quando un utente clicca il tasto di login, il modulo LoginWindow chiama il metodo action dell'EventListener e il MainWindow procede nello scambiare il modulo LoginWindow con il modulo FriendWindow e a caricare la Sidebar.

Il metodo action prevede anche un paramentro che viene usato per rendere più flessibile il sistema. Per esempio la Sidebar lo usa per dire alla MainWindow quale modulo caricare.

I moduli che presentano liste, come FriendWindow e ScoreboardWindow, hanno una funzionalità di scroll, in modo da permettere liste lunghe (la barra di scroll non è visibile, ma la rotellina del mouse funziona).

Istruzioni

Sia client che server usano la libreria JSONSimple vista a lezione, è quindi necessario aggiungerla tra le libreria aggiuntive in caso si usi un IDE. Né client né server richiedono parametri aggiuntivi. Assicurarsi sempre che la cartella data sia nel root del server.

Istruzioni per compilazione ed esecuzione su terminale per il server:

- Aprire il terminale nella cartella server
- Eseguire il commando *javac -classpath ".;../dependencies/json-simple-1.1.jar" *.java* su Windows o *javac -classpath ".:../dependecies/json-simple-1.1.jar" *.java* su Linux/MacOS per compilare
- Eseguire il *commando java -classpath ".;../dependencies/json-simple-1.1.jar" MainClass* su Windows o *java -classpath ".:../dependecies/json-simple-1.1.jar"* MainClass su Linux/MacOS per eseguire il server
- Per terminare il server, premere ctr+c. Il server salva i dati dopo ogni operazione importante, quindi non c'è rischio di perdita di dati.
- Aprire il terminale nella cartella client
- Eseguire il commando *javac -classpath ".;../dependencies/json-simple-1.1.jar" *.java* su Windows o *javac -classpath ".:../dependecies/json-simple-1.1.jar" *.java* su Linux/MacOS per compilare
- Eseguire il commando *java -classpath ".;../dependencies/json-simple-1.1.jar" MainClass* su Windows o *java -classpath ".:../dependecies/json-simple-1.1.jar" MainClass* su Linux/MacOS per eseguire il client
- Per chiudere il client, chiudere la finestra.

Possibili miglioramenti

Un possibile miglioramento sarebbe rendere la struttura del server ibrida, sfruttando sia socket non bloccanti con NIO che thread, in modo da rendere il server più efficiente e scalabile. Basterebbe creare un thread ogni N client e ogni thread gestice i suoi client attraverso selettori.

Altri due miglioramenti riguardano il client. Si potrebbe cambiare API per la gui, in modo da creare un'interfaccia più semplice da mantenere e più carina esteticamente (JavaFX è un buon candidato). Con il sistema corrente bisognerebbe spostare la business logic del client in un thread separato, in modo da non bloccare il thread della gui ogni volta che il client attende risposta del server.