TEXT TWIST

Implementazione del gioco text-twist multiplayer

Autore:
Emanuele Aurora

Progetto Reti dei Calcolatori e Laboratorio A.A. 2016/2017

Introduzione

Il progetto è suddiviso in 3 pacchetti: Client, Common e Server.

E' stata creata un'interfaccia grafica per il Client, e il Server non utilizza la non-blocking IO (NIO).
Oltre alle classi di questi pacchetti, sono presenti 4 File necessari per il funzionamento del progetto:

- ClientConfiguration.xml
  che contiene tutti i parametri necessari per l'inizializzazione del Client, come l'indirizzo IP del server.

- ServerConfiguration.xml
  che contiene tutti i parametri necessari per l'inizializzazione del Server, come le porte o il nome dei successivi due File.

- ServerState.xml
  che contiene lo stato del Server, cioè tutti gli utenti registrati e la classifica, dove ogni utente registrato è salvato con username e un hash della password.

- Words.txt
  che contiene la lista delle parole accettate dal server nel gioco.

Server

Il Server gestisce la maggioranza dei controlli interni al gioco, cioè quelli inerenti alla gestione degli utenti registrati, della classifica, della gestione delle partite in corso e del calcolo dei punteggi tramite l'uso del file delle parole.

Prima di entrare nei dettagli sul funzionamento del Server, adesso verranno discussi alcuni dettagli implementativi.

Gli utenti registrati vengono salvati nel file ServerState.xml, e nel farlo non viene direttamente salvata la loro password, ma invece è stato creato un hash di ogni password compreso di salt, che non permette a degli intrusi di accedere direttamente alle password. I dettagli su questo sistema verranno approfonditi in seguito.
Dopo averle lette dal file words.txt, le parole vengono caricate in memoria in un HashSet, per ridurre i tempi di verifica.

Nella seguente discussione, si introdurranno le classi usate mano a mano che viene spiegata la logica che è stata utilizzata nel Server.

- **Main**

la classe ServerMain gestisce l'inizializzazione delle variabili di sistema necessarie, e crea i thread che gestiranno le fasi successive di comunicazione.
Il primo compito è quello di aprire e leggere il file di configurazione per trovare tutti i dati necessari per la creazione dei ponti di collegamento, tra cui:

- le porte per le comunicazioni TCP, UDP, RMI e Multicast; quest'ultimo ha una porta iniziale, e questa viene incrementata per ogni connessione accettata in TCP, quindi esiste una porta unica per ogni gruppo multicast che deve ricevere i risultati di una partita.
  In realtà viene automaticamente creato un oggetto di tipo Multicast_rankings (classe che gestisce l'invio della classifica via multicast) ogni volta che il server riceve una richiesta di connessione TCP di qualunque tipo, anche una che non richieda l'invio della classifica, però questo non è dannoso dato che la chiamata del costruttore non è costosa e che un "buco" nelle porte usate non crea problemi (nel file di configurazione vengono definiti i limiti in cui può stare la porta di multicast, e ciclicamente resta entro questi).

- i nomi degli altri due file contenenti le parole per giocare e il file xml dello stato: cambiando questi valori si possono usare file con lo stesso scopo e un nome diverso.

Dopo aver costruito tutte le strutture dati necessarie, registra l'oggetto RMI, e avvia tramite una threadpool executor un server UDP che ciclerà tramite un thread di questa threadpool aspettando comunicazioni dal client (tramite UDP viene inviata la lista delle parole trovata dai giocatori in una partita).
Infine il thread stesso che ha chiamato il Main comincia un ciclo infinito per leggere le comunicazioni TCP e creare gli oggetti della classe ServerTask.

- **TCP comunication**

La comunicazione TCP è gestita interamente dalla classe ServerTask, i cui oggetti vengono creati dal main quando un utente apre una connessione tcp.
Questa classe è una delle più grandi del Server, dato che gestisce la creazione di nuove partite, l'accettazione e il rifiuto di partite già create, e l'invio della classifica globale.
La prima cosa che legge è il tipo di richiesta codificata con un numero, e poi semplicemente chiama la funzione che la gestisce.

Ogni comunicazione richiede un'iniziale scambio di dati per permettere al Server di capire chi sia il Client con cui sta comunicando. Oltre all'username, richiede un numero di identificazione (sessionID), che viene ricostruito ad ogni login, e inviato al Client per permettergli di fare richieste al Server. Questo numero aumenta leggermente il livello di sicurezza, dato che un'utente diverso da chi ha fatto il Login non può fare richieste a nome suo, però chiaramente mancando un layer di crifratura questo non è sufficiente per grandi applicazioni.
L'invio della classifica non lo richiede perché non è un metodo protetto.

Se il Server sta creando una nuova partita, alla fine della comunicazione col Client che ha creato la partita chiuderà la connessione e passerà allo stato di gestore di partita.
Il Client che ha creato la partita dovrà connettersi nuovamente per accettare la partita da lui stesso creato (viene fatto in automatico) e per avere un thread del Server che gestisca la sua ricezione delle parole della partita.
L'attesa del thread che gestisce la partita è implementata con una serie di sleep (ndr ammetto che si poteva implementare in modi molto più eleganti, ma il tempo è tiranno) di un secondo.
I thread di accettazione entrano nello stesso sleep, che dura 7 minuti come quello del thread principale, ma quasi certamente termineranno prima. I thread in questione comunicano con i due metodi readyGame() e isClosing() della classe Game che verrà discussa più avanti.

In pratica queste due variabili indicano quando tutti i Client hanno accettato l'invito (readyGame() == true) oppure per un problema o un rifiuto la partita è stata cancellata (isClosed() == true).
La sincronizzazione di questi metodi tra i vari thread del TCP è gestita direttamente nella classe Game.

Usando solo queste due informazioni, i thread possono sincronizzarsi e restare in attesa, per poi riattivarsi al momento giusto. Se tutto va a buon fine, il thread che gestisce la partita chiama il metodo play(), che al suo interno fa le cose standard per cominciare una partita, come trovare le lettere per la partita, e attendere che tutti i giocatori abbiano inviato le loro parole.

Se il Server sta leggendo una risposta di un Client, dopo la comunicazione in cui si accerta dell'identità e della partita, attende i 7 minuti sopracitati per l'inizio della partita e poi invia le parole decise dal thread che gestisce la partita. Infine chiude la connessione.

- **UDP, RMI e Multicast comunication**

Questi protocolli sono gestiti da UDP_task, RMI_server e Multicast_task rispettivamente.

UDP_task ha il semplice scopo di attendere i pacchetti inviati dai Client che all'interno hanno l'identificatore di sessione per assicurarsi che sia l'user corretto. Una volta letto un pacchetto, avverte i thread TCP semplicemente settando le parole lette nella partita (Game class).

L'RMI oltre a gestire le funzioni richieste, gestisce anche una richiesta del Client della lista degli utenti Online, così da poterla usare nell'interfaccia grafica durante la creazione di una nuova partita. Il registro RMI viene creato nel Main, e il Client deve solo fare una richiesta RMI al registro pubblicato nella porta contenuta nel suo file di configurazione.

Multicast ha il solo scopo di inviare la classifica, viene invocato dal thread creatore della partita (in TCP) quando questa è finita, e i Client possono ricevere i dati perché hanno ricevuto nelle informazioni dell'invito anche la porta multicast che poi sarà usata nell'invio della classifica.

Per inviarla, serializza un oggetto di tipo Multicast_rankings (nel common package, quindi a disposizione del Client) che poi verrà tradotto nella classifica dal Client.

- **Partite**

Il grosso delle informazioni delle partite è contenuto nella classe Game.
Questa, insieme a User e UserContainer, contiene il grosso della sincronizzazione del Server, anche se questo verrà discusso più avanti.

Per la gestione delle partite, ogni volta che ne viene creata una, è anche costruito un oggetto di tipo Game, che contiene tutte le informazioni della partita, come la classifica locale (quella della partita), la lista degli utenti partecipanti, il file caricato in memoria delle parole disponibili, in ID per identificare univocamente la partita e la lista delle parole scelte ed inviate dai singoli giocatori.
Per alcune variabili critiche sono state create Lock apposite, per una gestione più veloce, mentre quasi tutti i metodi che usano le altre variabili sono direttamente sincronizzati. Il metodo calculateRankings() non è sincronizzato perché è dato per scontato che sia l'ultima cosa chiamata dalla classe, quindi è compito di chi utilizza la classe tenerne conto.
In generale i metodi sono self-explanatory.

- **Stato**

Lo stato è gestito tramite la classe ServerState. Il suo scopo è quello di gestire il file ServerState.xml, di aprirlo alla creazione del Server e di crearne uno nel caso in cui manchi.

I metodi principali riguardano la creazione di un nuovo File, l'aggiornamento di questo (con l'aggiunta di un nuovo utente o l'aggiornamento del punteggio di uno già esistente) e la lettura da file per recuperare i dati salvati in questo.

Il file è in formato xml, ed è stato creato un formato per gestire lo stato dove:

- ogni utente è definito dal tag User,
- i tag contengono l'attributo ID che contiene l'username,
- il contenuto è definito dai tag Password, Salt e Points,
- per la Password e il Salt, sono stati definiti degli encoding per assicurarsi che il formato non incida sull'xml,
- i punti sono salvati come numeri.

Lo stato viene aggiornato (e il file salvato) ogni volta che un'utente si registra o termina una partita con l'aggiornamento dei punteggi.

- **Utenti**

per gestire gli utenti sono definite le classi User e UserContainer.

L'UserContainer è unico per ogni Server. Al suo interno contiene la lista degli utenti salvati (nel file di stato), delle partite create, e dei metodi generici per la gestione di questi.

Sono state usate strutture sincronizzate (LinkedBlockingQueue) per impedire conflitti dal multithreading. Questa classe contiene molti dei metodi sincronizzati del Server, dato che al suo interno ci sono le strutture dati che contengono utenti e partite.

L'User contiene i dati dei singoli utenti registrati, come l'username, l'hash della password e lo stato corrente del giocatore. E' stata creata la classe Passwords per semplificare la creazione di Hash nella fase di registrazione, e all'interno dell'User esistono due costruttori, uno per un nuovo giocatore (con una nuova password) e uno per il recupero di uno già esistente all'interno del file di stato. La crittografia scelta usa l'algoritmo SHA-1 e PBKDF2, e aggiunge alla password scelta il Salt prima citato per aumentarne la sicurezza.

- **Sincronizzazione**

La sincronizzazione è concentrata nelle tre classi che contengono le strutture dati, come già detto precedentemente, User, UserContainer e Game. La logica di fondo è che tutte le altre classi possono accedere indiscriminatamente a questi oggetti perché sono autonomamente sincronizzati, e quindi non è necessario preoccuparsi della sincronizzazione nelle classi che gestiscono la comunicazione.

Client

Il Client gestisce l'interfaccia grafica, e l'invio – lettura dei dati al Server per la gestione delle partite.

- La struttura del Client è piuttosto complessa, e mi scuso in anticipo per le dimensioni eccessive dalla classe GUI_logic. Mi è risultato semplificato costruire un'unica classe contenente la logica e delle classi innestate per i WorkerThread che gestiscono le comunicazioni col Server e le operazioni lunghe, così da poter sfruttare la visibilità degli attributi interni alla classe.

Per spiegare il funzionamento dell'interfaccia grafica, è necessario prima introdurre la logica di base per la gestione del multithreading.
E' stato seguito il protocollo standard del "single threaded GUI", cioè solo un thread può avere accesso all'interfaccia grafica e può modificare le sue componenti, dato che il multithreading applicato alle GUI è risaputamente complicato se non infattibile.
Vengono utilizzati 3 tipi di thread:

- Thread creatore, quello invocato dalla Main (ClientMain), il cui scopo è di invocare (InvokeLater) il thread gestore dell'interfaccia grafica,
- Thread Gestore dell'interfaccia, a cui è affidata ogni modifica di questa,
- Thread Worker, sono tanti thread creati ad hoc per fare determinati compiti che non possono essere affidati al thread worker per motivi di velocità; l'interfaccia non si deve bloccare (deve sempre essere sensibile alle azioni dell'utente) quindi il cose come operazioni lunghe o operazioni di attesa per le comunicazioni col server devono necessariamente essere gestite dai WorkerThread.

Dopo che la classe ClientMain ha letto i dati necessari dal file di configurazione ClientConfiguration.xml, e ha costruito le strutture dati necessarie, crea e avvia un'istanza di GUI_logic che identifica il Client. Per avviarlo usa java.awt.EventQueue.invokeLater, che da libreria di sistema dice al thread dell'interfaccia grafica di eseguire quel codice appena possibile.

- **Interfacce grafiche**

Il costruttore di GUI_logic successivamente avvia la prima interfaccia, quella di Login.
Per gestire la costruzione delle interfacce, è stato usato GroupLayer. Questo permette di definire in modo agevole e compatto tutto quello che serve.
Ogni pulsante di ogni interfaccia grafica è collegato ad un comando, definito come stringa costante all'inizio della classe, che è gestito dalla funzione actionPerformed().
Questa funzione viene invocata in automatica alla ricezione di un evento, e così è possibile leggere il codice di quel tipo di evento e chiamare la funzione che gestisce correttamente l'evento.
Il login è stato definito in modo da non permettere a due utenti di loggare contemporaneamente nello stesso account. Quindi, un secondo utente che proverà a farlo riceverà un messaggio d'errore dal Server.
Serve la richiesta di logout per settare un'account offline, e questa (gestita da RMI_server) necessita di username e password, così da non poter essere fatta da qualcun altro. Una chiusura inaspettata non invia la richiesta di logout, però il Server è comunque capace di verificare quando un Client è online tramite il metodo RMI isOnline(), che viene automaticamente chiamato quando un Client tenta il login. Tutto questo per impedire che dopo il crush inaspettato di un'utente questo non possa di nuovo effettuare il login.

Una volta effettuato il login, viene creata l'interfaccia di Lobby. Questa contiene le strutture necessarie per accettare le partite e per visualizzare la classifica globale. All'interno dell'interfaccia sono presenti numerosi controlli per impedire stati inconsistenti, per esempio non si può accettare una partita quando non se ne è selezionata nessuna.

Se si seleziona nuova partita, parte l'interfaccia per la costruzione di una nuova partita, NewGame.

Lobby e le successive interfacce di creazione di partite e di gioco sono costruite in modo che non ci siano inconsistenze su inviti tra giocatori innestati:

- se un'utente non è nella schermata di Lobby, ogni invito che questo riceverà verrà automaticamente rifiutato, e gli altri giocatori verranno avvisati,
- se un'utente crea una nuova partita, tutte le partite a cui è stato invitato verranno automaticamente annullate,
- se un'utente accetta una nuova partita, tutte le altre partite a cui è stato invitato verranno automaticamente rifiutate,
- quando una partita viene annullata, tutti gli utenti che avevano accettato la partita vengono avvisati tramite la chiusura della connessione TCP usata per ricevere le parole, e possono agire di conseguenza.

Tutti questi comportamenti sono gestiti tra il metodo tryToAccept() e la classe innestata acceptGameTask, che verrà spiegata meglio in seguito.

Dentro l'interfaccia di creazione di partita, si può selezionare i giocatori da invitare da una lista di giocatori online, ottenuta tramite la chiamata RMI di requestOnlineUsers(). L'interfaccia è fatta in modo da non poter creare una partita senza aver invitato nessuno, quindi deve esserci almeno un altro utente online per giocare.

La funzione valueChanged() è un eventHandler il cui scopo è impedire di premere su più liste contemporaneamente quando si invitano i giocatori. Viene usata anche per settare l'abilitazione dei bottoni ed impedire di creare inviti quando nessuno è selezionato.

Una volta selezionati gli utenti da invitare, si entra in un'interfaccia transitoria, che è uguale a quella di gioco, senza però permettere al giocatore di inserire parole da inviare. Questa interfaccia contiene un contatore che descrive il tempo limite dopo il quale la partita verrà annullata. Tutti i giocatori che accettano una partita (oltre a quello che l'ha creata) entrano in questa interfaccia e ci restano finché tutti hanno accettato o la partita è stata annullata.
E' stato usato un timer per la gestione dei contatori. Però tutti i contatori tranne quello dei 2 minuti di gioco sono solo di facciata: la partita non viene annullata dal Client anche dopo la fine dei 7 minuti di tempo, ma viene annullata direttamente dal Server quando il suo timer (che è quello valido) va oltre i 7 minuti. Quindi più Client che hanno accettato la stessa partita in momenti diversi, potrebbero avere un contatore diverso per i 7 minuti limite, ma questo non influisce sulla correttezza del comportamento.

Se tutti gli utenti accettano la partita, l'interfaccia transitoria si sblocca e viene visualizzata la parola scelta dal Server. Nei successivi due minuti gli utenti possono inviare parole dalle lettere visibili, ed è stato definito un controllo interno al Client per non inviare parole non valide (che non si possono formare dalle lettere del Server). Esiste comunque un doppio controllo anche da parte del Server per sicurezza.

Alla fine della partita viene visualizzata la classifica locale e segnalata la posizione di questa partita.

Se per vari motivi, la partita è iniziata, non può terminare finché il Server non lo decide. Quindi le "rese" degli utenti non inficiano sulla continuazione della partita, da dove poi tutti gli utenti rimasti in partita potranno vedere comunque i risultati. Questo implica che se tutti gli utenti si sono arresi, la partita continuerà ad esistere, e dopo la sua fine, anche se nessuno potrà vedere la classifica locale, quella globale verrà aggiornata di conseguenza.

- **Partite**

Le partite vengono salvate in un formato semplificato rispetto a quello del Server, infatti contengono solo lo stretto necessario da far vedere all'utente quando necessario, e alcuni dati per la lettura della classifica e l'identificazione della partita nel Server.
Sono state costruite le due classi GamesContainer e GamesData allo scopo, con i loro ovvi significati.

- **Metodi Helper**

Per la gestione degli eventi sulla pressione dei pulsanti, sono stati definiti numerosi metodi il cui scopo è definire il comportamento adatto ad ogni caso. Questi nella classe sono posizionati tra actionPerformed() e le classi innestate.
Il loro funzionamento è strettamente collegato con le classi innestate. Spesso il loro scopo sarà solo di settare alcune variabili e poi avviare un threadWorker che effettuerà il resto, come una comunicazione col Server. Non mi dilungherò particolarmente sulle funzioni dato che il loro scopo è già chiaro dal nome.
Il metodo tryToAccept() è particolarmente importante, perché sfrutta il modo in cui la classe innestata AcceptGameTask comunica col Server. Questa, permette di accettare un partita, ma anche di rifiutarne una, oppure di rifiutarne un gruppo quando si accetta una partita o se ne crea una.

- **Classi innestate**

Lo scopo delle classi innestate è chiaro a questo punto: comunicare col Server tramite TCP, RMI, UDP e multicast.
L'RMI è particolare, dato che è definito nel Main e il suo funzionamento semplicemente richiama la InvokeLater per far gestire la richiesta dal thread della GUI.
Gli altri tre fanno quanto richiesto dalle specifiche.
Per comprendere il funzionamento delle classi innestate basta conoscere il funzionamento dei WorkerThread, e dei metodi doInBackground() e done(), che sono affidati rispettivamente al worker e al thread della GUI.
I protocolli di comunicazione sono spiegati dentro le classi stesse e in ServerTask.

Consegna

La consegna del progetto contiene il progetto Eclipse contenente il codice sorgente, e una cartella contenente il formato .jar per eseguire direttamente il progetto compilato.
Per eseguire il codice si può usare direttamente i .jar dopo essersi spostati da terminale nella cartella che li contiene:

- In windows se è installato java tramite
  java -jar Server.jar
  java -jar Client.jar
  su terminali separati,
  oppure avviando direttamente prima il server e poi il Client.
- In Linux con
  java -jar Server.jar
  java -jar Client.jar
  sempre su terminali separati.