Server

Il server effettua il multiplexing dei canali mediante NIO. Le richieste vengono gestite con un Cached Thread Pool che rispetto al Fixed Thread Pool ha il vantaggio di occupare lo spazio in memoria dinamicamente in base al numero di richieste in elaborazione.

Il server è stato sviluppato mantenendo separate in package diversi le classi e i metodi che gestiscono i dati di WINSOME da quelle dedicate alla comunicazione con i client. Questo approccio ha reso più semplice lo sviluppo, la rilettura del codice e la correzione dei bug.

Le classi e i metodi che si occupano di memorizzare e manipolare i dati si trovano nel package "dataStructures", mentre la classe contenente il metodo Main e i Task dedicati alla comunicazione e alla gestione del ciclo di vita del Server si trovano nel package "server". Nel package "utilities" sono contenute alcune classi statiche usate per definire alcune semplici operazioni che sono state qui raccolte per evitare la duplicazione di metodi e anche per rendere il codice delle altre classi più corto e leggibile. Infine, nel package "gestioneRMI" sono definite classi e interfacce usate per il meccanismo di RMI.

Il Package **dataStructures**

Questo è il package dedicato alla gestione dei dati degli utenti, dei post e dei wallet associati agli utenti.

La classe principale di questo package è "StrutturaDatiServer" che ha come proprietà private:

- *autoriPost* che è una HashMap<String, ArrayList<Post>> usata per memorizzare tutti i post di ogni utente. La chiave, sfruttando la proprietà di univocità del nome utente scritta nelle specifiche, è il nome dell'utente autore dei Post.
  Scegliendo questa struttura dati, ho preferito dare priorità alle operazioni di lettura di potenzialmente grandi quantità di dati, rispetto alle operazioni di modifica ed eliminazione di un singolo post.
  Infatti, per riuscire a ricavare il nome dell'autore di un post dato l'id del post ho dovuto utilizzare la struttura dati ausiliaria descritta più avanti (indice).
  La classe *Post* usata nella hashMap rappresenta un post che oltre ad avere le proprietà da inviare ai client (idPost, autore, titolo, contenuto, lista di commenti, data) ha delle proprietà di supporto usate per alcune funzionalità del server, in particolare è da notare che il numero dell'iterazione del calcolatore di reward, il numero di like e dislike ricevuto dall'ultima iterazione e numero di commenti ricevuti dall'ultima iterazione sono usati per il calcolo periodico dei premi.
- L' *indice* è stato implementato come una lista di oggetti di tipo Tripla<P,OP,N>
  Dove:
  - P è l'id del post.
  - OP è l'id del post originale
  - N è il nome dell'autore

  Nel caso in cui un post è un rewin, P e OP saranno diversi. Ogni post che non sia un rewin ha P e OP uguali.
  In questo modo è possibile, partendo dall'id di un post, determinare tutti i suoi rewin e anche rewin di rewin a cascata. Questa proprietà è molto utile per l'eliminazione definitiva di un post e tutti i suoi rewin dal server.
- *Utenti* che è una lista di oggetti di tipo Utente che memorizza gli utenti registrati a WINSOME. La classe utente gestisce i follower, i following, e il wallet con la lista movimenti e il saldo.

I metodi in “dataStructures” sono dichiarati synchronized per permettere l’accesso concorrente agli oggetti, rendendo la struttura dati thread-safe.

Il Package **server**

Il server all’avvio legge il file di configurazione (config.json) e tenta la deserializzazione nella classe ConfigReader per mantenere in memoria i parametri scelti dall’amministratore del server. Se il file di configurazione è stato modificato in maniera errata e la deserializzazione non va a buon fine, il server viene comunque avviato con i valori di default. L’amministratore sullo standard output può verificare i parametri con cui il server è stato avviato.

Successivamente, il server tenta la lettura e la deserializzazione di eventuali dati salvati su file in una precedente esecuzione e li memorizza in un oggetto di tipo StrutturaDatiServer.

Viene poi inizializzata una lista per memorizzare gli utenti loggati; tale lista, essendo usata da più thread contemporaneamente è stata resa synchronized.

Una volta preparate le strutture dati per la gestione del server, vengono attivate le comunicazioni con i client:

- Inizializzazione di un registry RMI, e memorizzazione degli stub per l’operazione di iscirzione al server di un nuovo utente e la registrazione per le callback.
- Inizializzazione di un selettore per la comunicazione via TCP che si mette in ascolto sul channel relativo al socket del server, per ricevere nuove richieste di collegamento, e sui channel dei client collegati per ricevere le operazioni da effettuare.

Il server attiva alcuni thread di supporto:

- Uno che esegue il task definito in “TaskCalcolatoreGuadagni” per calcolare i reward e per inviare tramite multicast ai client registrati una notifica di avvenuto calcolo di premi.
- Uno che esegue il task definito in “TaskSalvataggioPeriodico” per il salvataggio periodico dei dati presenti sul server con lo scopo di evitare perdite indesiderate di informazioni in caso di errori fatali del server.

A questo punto, il server entra in un ciclo in cui gestisce le nuove richieste dei client.

Se arriva una richiesta di connessione, questa viene accettata e viene creato un nuovo channel per la comunicazione da e verso il client, e salvato nel selettore in modalità lettura

Se arriva da un client una stringa, questa viene sottomessa al pool di thread, che si occuperà di eseguire il task definito in “TaskServer”. Il task, dopo aver validato la richiesta, determina l’operazione da eseguire, e utilizza gli opportuni metodi offerti dalla classe “StrutturaDatiServer”. Al termine, invia un messaggio di risposta al client utilizzando il metodo della classe di supporto “Connection” che si trova all’interno del package *Utilities*.

Con questa gestione del pool di thread, il server mantiene attivo un thread solo per la durata dell’elaborazione di una richiesta. Questo garantisce migliori prestazioni, rispetto alla scelta di mantenere attivo un thread per ogni client connesso.

Il server esce dal ciclo quando un particolare client il cui nome utente è admin gli invia una speciale operazione di shoutdown.

Quando si verifica la condizione di uscita dal ciclo, il server inizia la procedura di chiusura: vengono chiuse le connessioni, terminato il pool di thread e inviate le interruzioni ai thread di supporto. Infine i dati vengono

serializzati e salvati su file per la successiva esecuzione sfruttando la classe di supporto GestioneGson contenuta nel package Utilities che sfrutta la libreria Gson.

Package:**utilities**
in questo package sono definite le classi di supporto che forniscono metodi utili al server. Tra queste, la più significativa è la classe *ConfigReader* usata all'attivazione del server per caricare i parametri di configurazione. La scelta di serializzare i parametri in formato json ha reso più semplice il parsing del file (essendo il file in formato standard il parsing può essere effettuato con il supporto di una libreria esterna) e automatica la validazione del formato dei dati.

In caso di errori nel file, il server viene comunque avviato con i parametri di default e viene generato un nuovo file di configurazione corretto.

Package:**gestioneRMI**

Contiene classi e interfacce utili per il corretto funzionamento dell'RMI. Il nome di questo package deve essere uguale al nome del package in cui vi sono le classi che si occupano dell'RMI lato client.
Oltre al metodo per la registrazione, richiesto nelle specifiche, e il metodo per iscriversi alle notifiche, indispensabile per il sistema di RMI/callback, il server mette a disposizione una funzione per ricavare la lista di followers di un determinato utente: metodo utile al client per richiedere la lista dei propri followers al momento del login.

Il server comunica l'indirizzo e la porta per il Multicast tramite RMI.

Client

Il Client all'avvio legge il file di configurazione, il quale come nel caso del server, è un file in formato json, deserializzato con la libreria Gson. L'uso di questo formato per il salvataggio della configurazione del client è utile perché l'operazione di parsing è semplicemente implementabile e quindi, volendo rendere il client ancora più "thin" si potrebbe addirittura eliminare la dipendenza con la libreria Gson.

Una volta letta la configurazione, il client apre la server socket per l'invio delle operazioni, mette in piedi il meccanismo RMI/callback per la ricezione di un nuovo follower, e attiva un thread che si occupa di registrarsi e ricevere i messaggi dal canale UDP del broadcast.

La lista che memorizza i followers di un utente è stata resa synchronized per risualtare thread-safe perché vi accede sia il thread attivato dal meccanismo di RMI (per aggiornare la lista follower) che il thread main (per mostrare la lista sullo standard output).

Il thread ausiliario attivato dal client si occupa della ricezione dei messaggi dal socket UDP.

Siccome l'operazione "receive" è bloccante, quando il thread è fermo su tale operazione non potrebbe essere interrotto. Per evitare questo problema, nel main viene inizializzato un oggetto di tipo "MulticastSocket", fondamentale per la comunicazione, che viene poi passato come parametro al thread: così facendo durante la chiusura del client, chiamo la close sul MulticastSocket forzando quindi la "accept" a lanciare una eccezione IOException permettando al thread di terminare.
L'indirizzo e la porta del multicast vengono ottenuti dal server tramite RMI.

Vista la richiesta di mantenere il client "thin", le operazioni richieste dall'utente vengono per la maggioranza inviate al server sul canale TCP senza effettuare alcun controllo.

Alcuni comandi però necessitano di un approccio differente:
-*register* utilizza la funzione del server messa a disposizione dal meccanismo di RMI per iscriversi al social

-*login*, il client invia la richiesta al server, quando riceve la risposta di avvenuto login richiede immediatamente la lista dei propri followers tramite un metodo RMI. Da questo momento in poi, la lista sarà aggiornata solamente dal server con il meccanismo di RMI/callback.
-*list followers*, non invia una richiesta al server, ma stampa la lista locale.
-*logout*, comunica al server la chiusura del channel e prepara il client alla chiusura.
-*shoutdown*, operazione eseguibile solo dall'utente con nome *admin*: impone la chiusura del server e del client.

Istruzioni per l'esecuzione

Il progetto usa come libreria esterna "gson.jar" sia nel client che nel server per la gestione di file di tipo json; entrambe le applicazioni non hanno bisogno di parametri da passare da linea di comando.
Le operazioni permesse al client sono quelle definite nel testo del progetto, con l'aggiunta del comando "shoutdown", il quale, se è invocato dall'admin(password: 000) comanda la chiusura del server.

Impostando nel file config.json la stringa "saved.json" il server si avvia con alcuni utenti e post precaricati, per permettere di testare le funzionalità del server.

Per compilare il client con javac bisogna usare il comando:

javac -cp .\gsonDependecy* *.java

per compilare il server con javac bisogna usare il comando:
javac -cp .\gsonDependecy* dataStructures*.java eccezioni*.java gestioneRMI*.java server*.java utilities*.java

I file JAR sono stati generati da Eclipse.