Introduzione

Il progetto è stato diviso in 3 packages: uno contenete il codice del client, uno il codice del server e l'ultimo contiene metodi "condivisi" accessibili da entrambi. Fa parte del progetto anche la cartella *recovery* che contiene le informazioni necessarie per ripristinare lo stato del server a seguito di un riavvio.

All'interno della cartella *recovery* sono presenti i progetti e l'elenco degli utenti registrati alla piattaforma. Questi ultimi sono contenuti all'interno del file *utentiRegistrati.json*, mentre ogni progetto ha una propria directory identificata dal nome del progetto: all'interno di ciascuna cartella è presente un file per ogni card (identificato dal nome della card) e un file contenente i membri del progetto (identificato con *members.json*). All'interno della cartella *recovery* sono già presenti dei file per testare il programma, ma è comunque gestita opportunamente l'eventualità in cui la cartella sia vuota. Nel caso in cui si volesse testare il programma a partire da questi dati, il server all'avvio stamperà l'elenco degli utenti e dei progetti presenti: per semplicità ho impostato la password di ciascun utente come *myPass*.

Server

Il server, all'avvio, ripristina lo stato del sistema leggendo dalla cartella *recovery* l'elenco degli utenti registrati e le informazioni dei vari progetti salvandoli in due strutture dati separate: *users* e *projects* che sono rispettivamente oggetti di tipo UsersDB e HashMap. Il primo implementa un'HashMap che associa ad ogni nome un utente, mentre il secondo è di per se un'HashMap che associa ad ogni nome un progetto. La scelta di utilizzare HashMap è dovuta all'efficienza della ricerca nella struttura dati.

Una volta che lo stato del sistema è stato ripristinato viene effettuata la stampa degli utenti registrati e dei progetti con i relativi membri.

Successivamente è possibile implementare la pubblicazione dei metodi remoti tramite un'istanza di tipo *RegistrationClass,* e *ServerNotificationService* per metodi remoti con callback, infine la connessione TCP viene implementata da un istanza di MultiThreadedServer. Successivamente verranno analizzate nello specifico.

Il server (ServerMainClass) dispone in oltre di una funzione che permette di salvare un file in memoria persistente a seguito di una modifica alla lista degli utenti o dei progetti, e una funzione che permette di generare un indirizzo IP univoco per ogni gruppo UDP-multicast nel range [239.0.0.0 – 239.255.255.255].

Client

Il client, all'avvio, inizializza le proprie strutture dati locali: una che contiene l'elenco degli utenti registrati alla piattaforma, l'altra mantiene l'associazione tra progetto e relativa chat mediante un'HashMap. Successivamente vengono "prelevati" i metodi condivisi dal server, viene effettuata l'iscrizione al sistema di notifiche (condividendo la funzione di aggiornamento) e instaurata la connessione TCP col server.

Il client legge richieste da standard input e a seconda della richiesta decide se eseguirla in locale o farla gestire al server:

le operazioni eseguite in locale sono *listUsers* e *listOnlineUsers* che vanno a leggere la struttura dati locale degli utenti; le funzioni di lettura e scrittura della chat sono anch'esse implementate in locale: dopo aver effettuato i controlli di sicurezza, si va a recuperare la chat del progetto salvata nella struttura dati locale *chats* e, a seconda dell'operazione, si invia un messaggio al gruppo multicast o si recuperano i messaggi non letti. Tutte le altre operazioni sono gestite direttamente dal server ad eccezione della funzione *register* che va ad invocare il metodo remoto messo a disposizione dal server.

RMI- Registrazione alla piattaforma

La registrazione di un utente alla piattaforma è gestita tramite il meccanismo RMI. Il server dichiara un'istanza di *RegistrationClass* che chiama a sua volta il metodo *start*, il cui scopo è quello di "pubblicare" la funzione all'interno di un registro creato sulla porta 4567; il client inizialmente andrà a "prelevare" questo metodo remoto per poi invocarlo su richiesta. L'invocazione di questo metodo da parte del client permette di aggiungere una nuova entry alla struttura dati *users* (del server): sarà il server ad effettuare i controlli di esistenza e in caso aggiornare la struttura dai *users* e comunicare tale modifica agli altri utenti tramite il sistema di notifica. Quindi qualsiasi modifica da parte degli utenti verrà effettuata direttamente alla struttura dati del server. Successivamente la modifica verrà salvata in memoria persistente tramite la funzione *saveFile*.

RMI Callbacks- Aggiornamento strutture dati

Per prima cosa il server pubblica, mediante *NotificationClass.start*, i metodi che permettono all'utente di iscriversi e cancellarsi al sistema di notifica. Questo metodo ritorna un'istanza di *ServerNotificationClass* che è la classe vera e propria che implementa sia le funzioni di iscrizione e cancellazione alle callback sia la procedura di notifica *update*. Dall'altra parte, il client, ha il compito di recuperare i metodi pubblicati dal server e a sua volta di pubblicare il metodo *notifyEvent* che il server invocherà per mandare la notifica ad un singolo cliente.

In breve quando il server invocherà la funzione *update*, per ogni client iscritto verrà invocata la funzione *NotifyEvent* che ha il compito di prendere come parametro la struttura dati aggiornata e sostituirla alla struttura dati locale del client: in questo modo ogni client avrà una struttura dati sempre aggiornata.

La funzione *update* verrà invocata dal server ogni qual volta viene modificata la struttura dati *users*: ogni volta che un nuovo utente si registra alla piattaforma, che effettua il login o che effettua il logout. Grazie alla struttura dati sempre aggiornata tramite le callbacks, l'utente può eseguire in locale la richiesta di visualizzare la lista di tutti gli utenti e la lista degli utenti online in locale senza doverla inoltrare al server.

TCP- Gestione richieste

Siccome un server TCP dev'essere in grado di gestire richieste provenienti da più client contemporaneamente, ho deciso di implementarlo multithreaded gestendo opportunamente la concorrenza alle strutture dati condivise dai vari thread: *users* e *projects*, rispettivamente la lista degli utenti iscritti e la lista dei progetti creati. Per far si che il server sia in grado di gestire i thread in modo autonomo ho implementato un threadpool.

Sulla connessione TCP avviene lo scambio principale di messaggi tra client e server: il client si collega alla porta 4569 su cui è in ascolto il server e la comunicazione avviene secondo il paradigma richiesta-risposta. Ogni qual volta che il server accetta una richiesta di connessione da un client, viene creato un task di tipo *RequestHandler* e passato in gestione al threadpool.

I thread del threadpool hanno il compito di leggere la richiesta del client, eseguirla e infine comunicare l'esito: per ogni richiesta vengono effettuati dei controlli sull'esistenza dell'oggetto richiesto e di accesso da parte dell'utente a quell'oggetto, ad esempio se un utente che vuole eliminare un progetto è necessario che sia effettivamente membro del progetto e che quel progetto esista.

Ogni funzione richiesta dal client ha un proprio handler all'interno del task, nel caso in cui il server riceva una funzione sconosciuta restituirà un messaggio con l'elenco delle funzioni disponibili. Ogni qual volta si effettuano modifiche allo stato del sistema (nuovo utente registrato, nuovo progetto creato, …), queste modifiche vengono salvate in memoria persistente tramite la funzione *saveFile* presente in ServerMainClass. Quando un utente si disconnette dal sistema, il server cattura l'eccezione (IOException) e ha il compito di aggiornare la struttura dati (settando lo stato dell'utente a offline) e comunicarlo agli altri utenti.

In quanto server multithreaded, è necessario gestire la concorrenza nell'accesso alle risorse condivise da parte dei vari thread: gli oggetti condivisi, come detto prima, sono *users* e *projects*; tuttavia le lock da acquisire sono 3 di tre tipi: una per accedere in mutua esclusione alla lista degli utenti, una per accedere alla lista dei progetti che generalmente è mantenuta soltanto per la ricerca di un progetto nella lista. Una volta trovato, questa lock viene rilasciata e acquisita sul singolo progetto: in questo modo quando si opera su un singolo progetto non si impedisce l'accesso a tutti gli altri.

All'interno della connessione TCP vengono inviati anche indirizzo IP e porta della chat di un progetto, a seguito dell'invio da parte del client della richiesta *joinChat*. Ho preferito passare la coppia [indirizzo IP multicast, porta] di ciascuna chat su richiesta poiché non è detto che un client, al momento del login, voglia stare in ascolto su tutte le chat di cui fa parte ma solo su alcune, facendo risparmiare risorse computazionali al client che, in quanto "utente generico" potrebbe trovarsi su macchine con risorse molto limitate.

UPD Multicast - Chat

Le chat sono implementate come gruppi multicast, dove i membri di un progetto restano in ascolto, inviano messaggi o leggono quelli non letti. Ogni progetto ha una propria chat, ciò vuol dire che ad ogni progetto corrisponde un indirizzo IP multicast scelto dal server tramite la funzione *generateIP*. Ogni utente deve avere la possibilità

di accedere alle chat dei progetti di cui fa parte per inviare e ricevere messaggi, ma siccome il client per accedere a queste chat deve conoscere l'indirizzo IP multicast, è necessario che il server glielo comunichi in qualche modo: a seguito di una richiesta di partecipazione alla chat (*joinChat*). Quando un utente vuole unirsi alla chat di un progetto manda la richiesta *joinChat* al server (sulla connessione TCP), il quale risponderà con IP e porta del relativo gruppo multicast. A questo punto il client avvia un thread *Chat* che resta in ascolto di messaggi e li salva temporaneamente in una coda del thread. L'invio e la ricezione di messaggi sono funzioni definite all'interno della classe *Chat* e hanno il compito rispettivamente di andare a scrivere nel gruppo multicast e di leggere i messaggi nella coda dei messaggi non letti.

Il client una volta che si unisce ad una chat, salva all'interno di un'HashMap la corrispondenza tra nome progetto e chat: così nel caso di invio o ricezione di messaggi, andrà ad inviare ad un indirizzo multicast già noto o a leggere da una struttura dati facilmente recuperabile.

Il server può mandare un messaggio sul gruppo multicast solo per due possibili motivi: una card viene spostata o il progetto viene eliminato. Quest'ultima è stata una scelta voluta in modo tale che i thread client in ascolto, alla ricezione di un messaggio di chiusura ("*system: close"*), capiscano che il progetto è stato eliminato e la chat dev'essere chiusa: quindi ciascun thread deve terminare l'ascolto e settare il flag *Is_Listening* a false. Infine quando un client richiederà l'accesso ad una chat (per lettura o scrittura), verrà controllato questo flag che stabilirà se eseguire l'operazione o eliminare l'associazione nome_progetto-chat stampando successivamente un messaggio di errore.

Istruzioni

Compilazione: `javac ./client/*.java ./common/*.java ./server/*.java`

Esecuzione Server: `java server.ServerMainClass`

Esecuzione Client: `java client.ClientMainClass`

Operazioni

- `register username password` -un utente si registra alla piattaforma
- `login username password` -un utente si connette
- `logout username` -un utente si disconnette
- `listUsers` -visualizza la lista degli utenti registrati
- `listOnlineUsers` -visualizza la lista degli utenti attualmente online
- `listProjects` -visualizza la lista dei progetti di cui l'utente fa parte
- `createProject projectName` -crea un nuovo progetto
- `addMember projectName newMember` -aggiunge un nuovo membro al progetto
- `showMembers projectName` -visualizza i membri di un progetto
- `showCards projectName` -visualizza tutte le card di un progetto
- `addCard projectName cardName description` -aggiunge una nuova card al progetto
- `moveCard projectName cardName listaPartenza listaDestinazione` -sposta una card di un progetto da una lista ad un'altra
- `getCardHistory projectName cardName` -visualizza la cronologia dei movimenti di una card
- `joinChat projectName` -permette ad un utente di entrare nella chat di un progetto
- `readChat projectName` -visualizza i messaggi non letti all'interno della chat
- `sendChat projectName message` -invia un messaggio nella chat del progetto
- `cancelProject projectName` -elimina un progetto
- `close` -termina il programma client
- qualsiasi altro comando restituisce un messaggio di aiuto contenente i comandi disponibili.