Progetto WORTH: WORkTHoghter

1 Descrizione del problema (da specifiche)

Negli ultimi anni sono state create numerose applicazioni collaborative, per la condivisione di contenuti, messaggistica, videoconferenza, gestione di progetti, ecc. In questo progetto didattico, WORTH (WORkTogetHer), ci focalizzeremo sull'organizzazione e la gestione di progetti in modo collaborativo. Le applicazioni di collaborazione e project management (es. Trello , Asana) aiutano le persone a organizzarsi e coordinarsi nello svolgimento di progetti comuni. Questi possono essere progetti professionali, o in generale qualsiasi attività possa essere organizzata in una serie di compiti (es. to do list) che sono svolti da membri di un gruppo: le applicazioni di interesse sono di diverso tipo, si pensi alla organizzazione di un progetto di sviluppo software con i colleghi del team di sviluppo, ma anche all'organizzazione di una festa con un gruppo di amici.
Alcuni di questi tool (es. Trello) implementano il metodo Kanban (cartello o cartellone pubblicitario, in giapponese), un metodo di gestione "agile". La lavagna Kanban fornisce una vista di insieme delle attività e ne visualizza l'evoluzione, ad esempio dalla creazione e il successivo progresso fino al completamento, dopo che è stata superata con successo la fase di revisione. Una persona del gruppo di lavoro può prendere in carico un'attività quando ne ha la possibilità, spostando l'attività sulla lavagna.
Il progetto consiste nell'implementazione di WORkTogetHer (WORTH) : uno strumento per la gestione di progetti collaborativi che si ispira ad alcuni principi della metodologia Kanban.

2 Strutturazione del 'file system' del progetto

All'interno della directory *WORTH* troviamo altre due directory: *src* e *Data*. La seconda servirà per memorizzare, in formato .json, i dati relativi ai progetti e agli utenti del nostro servizio (rispettivamente nelle cartelle *ProjectsData* e *UsersData*), mentre la prima contiene tutti i file .java fondamentali per l'esecuzione del nostro progetto.
In particolare ci saranno i due file main per far partire l'esecuzione del client e del server e tre directory importanti:

- *Client*, per tutti i file .java che si occupano di definire aspetti del client
- *ServerWorth*, per i file .java che definiscono i comportamenti del server
- *ExtLibraries*, per le librerie esterne di jackson, fondamentali per la serializzazione e quindi la persistenza del sistema
- *MyExceptions*, per le eccezioni create da me

3 Utilizzo del programma

Per iniziare a usare WORTH bisogna prima di tutto compilare i file .java.
Per far partire l'esecuzione bisogna spostarsi nella cartella 'src' del progetto il cui path è: ***/WORTH/src***
Successivamente si procede alla compilazione dei file .java:
javac -cp ExtLibraries/* *.java MyExceptions/*.java Client/*.java ServerWorth/*.java
A questo punto il programma è pronto per essere eseguito. Sarà necessario mandare in esecuzione il server per primo con il comando da terminale:
java -cp .:ExtLibraries/jackson-annotations-2.12.1.jar:ExtLibraries/jackson-core-2.12.1.jar:ExtLibraries/jackson-databind-2.12.1.jar MainClassServer
Avviato il server, si può procedere con l'esecuzione del client tramite il comando:
java -cp .:ExtLibraries/jackson-annotations-2.12.1.jar:ExtLibraries/jackson-core-2.12.1.jar:ExtLibraries/jackson-databind-2.12.1.jar MainClassClient [optional server IP]
(Nota: opzionalmente si può specificare l'IP del server quando si manda in esecuzione il client. Per recuperarlo si può guardare la stampa sul terminale del server).
Fatto ciò il programma è ufficialmente funzionante e si può iniziare a lavorare con WORTH.

4 Analisi del progetto: alcuni appunti sulle classi e analisi delle scelte

4.1 Le classi MainClient e MainServer

Sono le due classi che fanno partire rispettivamente l'esecuzione del Client e del Server. Per quanto riguarda il Server, il main istanzia semplicemente un oggetto di tipo WorthDB (vedi sotto); per il Client si controlla invece se è presente un'altro IP per il server a cui deve connettersi, altrimente si assume che il server sia in esecuzione sul localhost. A quel punto si crea un thread con un oggetto di tipo ClientTasks (vedi sotto) e si manda in esecuzione.
Tutte le restanti azioni sono gestite dalle classi che analizzeremo più avanti.

4.2 Card, Progetto e Utente

Queste classi sono quelle che permettono la definizione degli oggetti principali del progetto e del servizio.
La classe Utente permette la definizione di oggetti che modellano le caratteristiche di un utente WORTH:

- username è la variabile legata al nome con cui l'utente vuole farsi chiamare nel servizio
- passw è la variabile legata alla password con cui l'utente accede e si registra
- stato è la variabile che indica se l'utente è online o offline

Inoltre, ho definito alcuni metodi Set e Get per il recupero o la definizione dei valori di queste variabili. Si può notare anche un secondo costruttore utilizzato da jackson.
La classe Card è la classe che consente di modellare le card del progetto, ovvero dei piccoli compiti definiti dagli utenti membri del progetto e che vanno completati.
Possiamo notare:

- name, nome della card a cui si fa riferimento
- description, descrizione testuale della card
- history, una LinkedList<String> che contiene tutti gli spostamenti di una card (ovvero tutte le liste di lavoro da lei visitate)

Anche qui si possono notare alcuni metodi Get e Set (Nota: l'annotazione @JsonIgnore specifica a jackson di non serializzare). Anche in questa classe è presente un costruttore per jackson, con la differenza che in questo caso il costruttore inizializza le variabili a null invece di chiamare super().

La classe Progetto, infine, è la classe che permette di modellare un progetto creato da un utente del servizio:

- projectName è il nome del progetto
- creator è il creatore del progetto
- chatIP è la variabile legata all'indirizzo IP Multicast della chat di progetto
- memberList è una LinkedList<String> che tiene traccia dei membri del progetto
- trackingCards è una lista di Card che tiene traccia di tutte le card del progetto; ho deciso di usare una struttura dati per il tracciamento di tutte le card per semplicità e immediatezza, senza dover fare un metodo apposito
- infine ci sono le quattro liste di lavoro, tutte contenenti oggetti di tipo Card

Anche in questo caso è presente un costruttore per jackson che inizializza le liste e assegna il valore 'null' a tutte le altre variabili.
Tra i metodi, oltre ai soliti metodi Get e Set, possiamo notare getCard (per recuperare tutte le card del progetto), getListByName (per recuperare le liste di lavoro) e addAllCards (usato per inserire le card nelle varie liste dopo la deserializzazione).

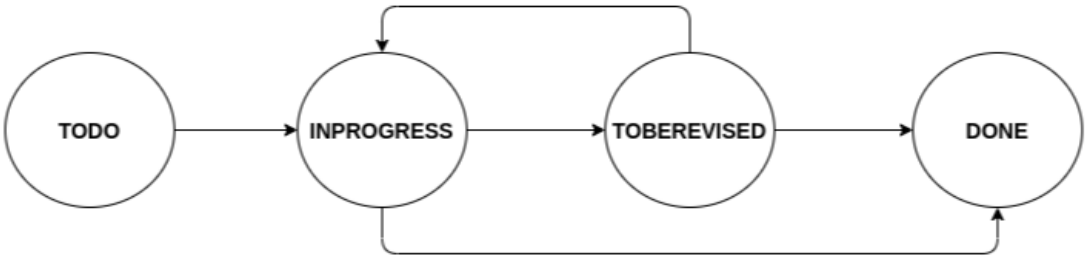
4.3 WorthDB e RMIServerInterface

RMIServerInterface è un'interfaccia per modellare la registrazione ai servizi offerti da WORTH e definisce delle operazioni che utilizzano RMI per dialogare con il client.

WorthDB è, invece, la classe che modella il server WORTH, definendolo come un database. In particolare, è da notare come questa classe si occupi di quasi l'intero setup del server: innanzitutto carica le liste relative agli utenti e ai progetti con i dati memorizzati tramite la serializzazione e rende le operazioni su di esse synchronized grazie a 'Collections.synchronizedList', al fine di garantire la concorrenza. Inoltre, sempre in questa classe, vengono pubblicate le risorse RMI che permetteranno al client di eseguire l'operazione di registrazione e successivamente il server viene messo in attesa di connessioni tramite la dichiarazione di una variabile di tipo

TCPManagement. Possiamo quindi notare come questa classe sia fondamentale per il corretto funzionamento di WORTH.
Questo anche perché i metodi definiti all'interno di questa classe forniscono le risposte alle richieste del client tramite TCP o RMI.
Di seguito lascio una breve analisi di due metodi (che non fanno parte delle operazioni del client ma che vengono usati per fornire le risposte corrette) per spiegare meglio come funzionano e a cosa servano:

- checkList, serve per controllare che gli spostamenti di una card da una lista di lavoro a un'altra rispettino le specifiche, ovvero il seguente schema:



- createChatIP, serve per creare l'indirizzo IP di una chat di progetto. Poiché gli indirizzi IP Multicast (le specifiche richiedevano indirizzi di questo tipo) vanno da *224.0.0.0 a 239.255.255.255,* per definire il primo byte scelgo casualmente un numero tra 224 e 239; successivamente, scelgo di nuovo randomicamente i tre byte successivi, ma rendo l'indirizzo IP unico andando a sommare al numero randomico generato (tra 0 e 255) il valore dell'indice (all'interno della lista) del progetto di cui voglio l'IP della chat; fatto ciò, calcolo il numero in modulo 255 (% 255) per essere sicuro che l'indirizzo IP finale sia valido.

4.5 TCPManagement

Questa classe è quella responsabile del corretto funzionamento della connessione TCP tra il Server e il Client.
La soluzione scelta per una corretta implementazione è stata quella di utilizzare il Multiplexing dei canali tramite selettore. Ho fatto questo tipo di scelta in quanto utilizzare Multithreading o un ThreadPool per la gestione delle connessioni introdurrebbe problemi di scalabilità e di efficienza (considerato che il numero di client non è definito), mentre il Multiplexing gestisce più connessioni con un solo thread (il selettore decide quali canali sono pronti per la lettura o la scrittura), riducendo i problemi di overhead e migliorando performance e scalabilità.

Come possiamo notare, infatti, il selettore itera sulle SelectionKey (riferimenti ai vari canali) e controlla quali sono pronte per quali operazioni (successivamente gestite da manageAccept, manageRead e manageWrite).

Importante è anche il controllo sulla disconnessione del client: nel caso non avesse effettuato il logout, viene forzato. In seguito si cancella la key e si chiude il canale associato.
Una breve analisi di manageRead è lasciata di seguito: si tratta del metodo che gestisce la lettura sul canale del client. Dopo aver letto, tramite buffer, la richiesta, controlla il label (etichetta) che specifica l'operazione richiesta dal client. A quel punto a seconda del label viene invocato uno dei metodi definiti in WorthDB per la gestione delle operazioni e, terminato, si ritorna una risposta al client.

4.6 DataSerialization

Questa classe viene utilizzata per definire i meccanismi di serializzazione e deserializzazione dei dati. Inizialmente crea le directory specifiche che contengono i file .json tramite dei paths predefiniti (se queste non esistono). I metodi definiti hanno il compito di memorizzare e caricare solo i dati importanti degli utenti, dei progetti e delle cards.
In particolare i campi serializzati sono:

- nome, password e stato per l'utente (lo stato è stato necessario serializzarlo per evitare inconsistenza con la stampa degli utenti online)
- nome, creatore, chat IP, memberList e cardTracking per il progetto (chat IP per evitare di mantenere una lista nel database e perché la creazione dell'IP è randomica)
- nome, description e history per le cards

4.7 Chat e MSGHandler

Queste due classi sono utilizzate per la gestione dei messaggi e della chat. La seconda manda in esecuzione un thread che rimane in attesa di messaggi sulla chat e li mostra al client su richiesta. La prima classe è invece quella che definisce la chat vera e proprio e le operazioni che verranno poi usate dal client per comunicare e leggere.

4.8 ClientInterface e ClientTasks

La prima è un'interfaccia che definisce alcune delle operazioni eseguibili dal client. Importante è il metodo notifyEvent che aggiorna le strutture dati locali del client in seguito a una callback (dovuta a una modifica nello stato di un altro utente).
La seconda è la classe che definisce il client (come thread) e che permette al client di connettersi al server sia tramite TCP che tramite RMI.
Per prima cosa vengono appunto eseguiti i setup della connessione TCP e dell'RMI, dopodiché si definisce un lettore per cattura l'input del client da linea di comando. Da quel punto in poi, fino a che il thread è attivo, viene letto il comando del client, parsato e viene inoltrata la richiesta al server.

Si noti che nello smistamento della richiesta ho usato due implementazioni diverse: una basata su if-else per quando i comandi possibili erano pochi e una basata su

switch per quando i comandi possibili erano troppi da poter essere gestiti facilmente con if-else.

Metodo degno di ulteriore analisi in questa classe è askToServer, utilizzato per comunicare con il Server e inoltrargli la richiesta, per poi attendere una risposta da quest'ultimo. Gli altri metodi sono principalmente legati alle operazioni di WORTH e non fanno altro che definire il label di identificazione prima di comunicare con il server.

5 Strutture utilizzate

Come già detto, le strutture principali sono relative alle liste usate in WorthDB e che vengono ottenute tramite l'invocazione del metodo Collections.synhronizedList, per poter gestire la concorrenza sulle operazioni 'native' di queste strutture dati. Tutte le altre operazioni che operano su queste liste (login, logout, register e i metodi per le callbacks di RMI) sono stati definiti synchronized in modo che possano pure loro essere adattati a un ambiente concorrente.
Per quanto riguarda TCP, abbiamo già mostrato come sia stato scelto di utilizzare il Multiplexing dei canali, per tanto sarà il selettore a occuparsi di gestire una richiesta per volta.
Si noti che RMI e TCP gestiscono richieste differenti: in particolare ad RMI è affidata la registrazione e tutte le operazioni di callback, mentre TCP si occupa delle richieste rimanenti.

6 Schema dei Threads

Client
I threads lato client sono relativi alla classe ClientTasks, i cui threads vengono mandati in esecuzione dal main ogni volta che un nuovo client si connette, e alla classe MSGHandler, che manda in esecuzione un handler per gestire i messaggi sulla chat (ovviamente c'è da considerare anche il main thread del client).
Server
Il thread lato server riguarda solo ed esclusivamente il main thread.

7 Le eccezioni

Nella cartella MyExceptions sono contenute le eccezioni create da me per identificare casi particolari di errore:

- AlreadyLoggedException, se l'utente tenta di accedere nuovamente
- WrongPasswordException, se la password è sbagliata
- UserNotExistsException, se l'utente non esiste
- NotMemberException, se l'utente non è un membro del progetto
- ProjectExistingException, se esiste un progetto con lo stesso nome
- AlreadyMemberException, se l'utente è già membro
- NotRegisteredException, se l'utente non è registrato

8 Sintassi dei comandi

Sebbene sia presente un comando ‘help’ per la stampa della sintassi dei comandi, la riporto pure qui per maggiore comprensione:

- list_users -> per la stampa di tutti gli utenti registrati
- list_online_users -> per la stampa di tutti gli utenti online
- list_projects -> per la stampa dei progetti di cui un utente è membro
- create_project projectName -> per creare un progetto
- open_project projectName -> per accedere a un progetto
- add_member usernameToAdd -> per aggiungere un membro
- show_members -> per mostrare tutti i membri di un progetto
- show_cards -> per mostrare tutte le cards di un progetto
- show_card cardName -> per i dettagli di una card
- add_card cardName description -> per creare una nuova card
- move_card cardName source dest -> per spostare una card da source a dest
- get_card_history cardName -> per sapere tutti gli spostamenti di una card
- cancel_project -> per eliminare un progetto
- read_chat -> per leggere la chat
- send_message message -> per inviare un messaggio
- logout -> per terminare

Nota: è importante che l’underscore venga mantenuto, per evitare di incorrere in errori. I nomi delle liste sono TODO, IN_PROGRESS, TO_BE_REVISED e DONE: non è importante che siano in maiuscolo, ma anche qui è importante mantenere l’underscore.