Laboratorio di Reti - Corso A

WORTH: WORkTogetHer

Docenti: Federica Paganelli, Laura Ricci
Studente: Ruslan Stasula, Matricola 580618

UNIVERSITÀ DI PISA

Dipartimento di Informatica
05-01-2021

Indice

1 Introduzione

Il progetto riguarda la programmazione di un tool di organizzazione e gestione dei vari task "AGILE", secondo il metodo Kanban, per aiutare gli utenti a organizzarsi e coordinarsi nello svolgimento di progetti comuni, utilizzando il linguaggio Java. 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. Un insieme di attività, definite da card in questo caso, compongono un progetto. E' richiesto anche di gestire una chat per ciascun progetto attivo sulla piattaforma, in cui il server invia notifiche di spostamento delle card ai vari utenti membri, e gli stessi partecipanti comunicano tra loro. La registrazione degli utenti al servizio avviene tramite Remote Method Invocation, mentre il dialogo tra client e server secondo connessione TCP. La chat, invece, sfrutta il protocollo UDP, in particolare fa uso dei MulticastSocket di Java. Per la gestione dei file di persistenza, contenenti le informazioni delle strutture dati del server, è stato utilizzato il formato JSON con l'aiuto della libreria Gson (Google).

2 Istruzioni per Compilazione e Avvio

Il progetto è stato testato su ambienti Linux (Arch) e Windows 10 con JDK 11.

- Compilazione ed esecuzione Server:

  Compilazione:

  ```
  Spostarsi all'interno della directory WORTH
  javac -cp "./gson/gson-2.6.2.jar:" -d ./out -sourcepath ./src ./src/Server.java
  ```

  Esecuzione:

  ```
  Nella directory WORTH
  java -cp "./gson/gson-2.6.2.jar:./out" Server
  ```

- Compilazione ed esecuzione Client:

  Compilazione:

  ```
  Spostarsi all'interno della directory WORTH
  javac -cp "./gson/gson-2.6.2.jar:" -d ./out -sourcepath ./src ./src/Client.java
  ```

  Esecuzione:

  ```
  Nella directory WORTH
  java -cp "./gson/gson-2.6.2.jar:./out" Client
  ```

3 WORTH Server

Il server appena avviato, crea gli oggetti necessari per la gestione dei dati (classi ProjectsManager e UsersManager, anche per la persistenza), configura l'RMI per le iscrizioni degli utenti, ed avvia il ThreadPool che si occuperà della gestione delle richieste da parte dei vari client collegati (attraverso il ClientTask). Per ultima cosa il server si mette in ascolto sulla sua ServerSocket, accettando le connessioni TCP e gestendo le successive richieste con nuovi thread.

3.1 Threads e concorrenza

Essendo il server multi-threaded, c'è bisogno di regolare gli accessi alle strutture dati della parte che gestisce gli utenti, e quella che gestisce i progetti, che saranno presentate più avanti. Per questo scopo, i metodo di accesso e modifica di tali strutture sono **synchronized**, pertanto si ricorre al monitor dell'oggeto con la lock intrinseca per l'accesso a tali strutture dati.

3.2 ClientTask

Questa entità non è altro che il Runnable passato in "pasto" al ThreadPool. Il suo compito, dopo aver configurato gli stream di comunicazione TCP con il client, è quello di ciclare, leggendo di volta in volta la stringa-comando che arriva dal client (sul BufferedReader, trattandosi solo di stringhe...) , interpretarla e, se è un comando corretto, che il server quindi riconosce, lo esegue, restituendo il corrispettivo codice del risultato (e, nel caso, l'annessa struttra dati) dell'esecuzione sull'ObjectOutputStream. Se viene intercettata un'eccezione riguardante in qualche modo la connessione e la comunicazione del server con il client, il thread esegue un logout forzato dell'utente dal sistema e termina, in quanto non c'è più connessione. Altrimenti ClientTask continua a ciclare, fino a che l'utente non esegue il logout. Tuttavia, chi fa il vero lavoro sporco, sono UsersManager e ProjectsManager, che appunto implementano tutta la parte di gestione degli utenti e dei progetti.

3.3 UsersManager

Questa entità implementa tutte le operazioni che riguardano gli utenti del servizio e le loro credenziali, compresa quindi anche la registrazione (RMI).
NB. Tutte le password degli utenti vengono HASH-ate.

3.3.1 Dati

- **private HashMap<String, String> passwords**
  private HashMap<String, User> users
  Memorizzano rispettivamente le password degli utenti, con username come chiave, e gli utenti, con il loro stato, con stessa chiave.

- **private final ArrayList<CallbacksInfo> clientCallbacks**
  Struttura che memorizza le informazioni necessarie per le Callback ai client.

3.3.2 Metodi

- **regUser** : metodo utilizzato per la registrazione degli utenti. Si controlla che l'utente con il nome fornito non sia già registrato e lo si inserisce nella dovuta struttura dati, aggiornando anche il file JSON di persistenza.
- **usrLogin** : metodo usato per l'autenticazione degli utenti. Si controlla che l'utente sia registrato al servizio, e poi si controlla la sua password, restituendo come risultato una tripla di valori (classe Triplet) in cui si restituisce al client come primo valore il codice del risultato dell'operazione, che eventualmente segnalerà il tipo dell'errore. Il secondo valore ritornato è la lista degli utenti del servizio, con il loro stato (online/offline), mentre il terzo è la lista contenente le informazioni per le chat dei progetti di cui l'utente è membro.
- **usrLogout** : metodo che serve per il logout di un utente dal servizio. Restituisce soltanto un codice di risultato.
- **hashPw** : questo metodo effettua l'hashing delle password che gli utenti immettono alla registrazione, o al login. Si istanzia un oggetto MessageDigest, con l'algoritmo "SHA-256" e si produce una rappresentazione della password sotto formato di stringa esadecimale.
- **checkUser** : controlla che un utente appartenga all'insieme di quelli registrati.
- **updatePwJSON** : scrive sul file JSON di permanenza l'HashMap che mappa uno username all sua password.
- **updateOnlineUsers** : metodo che sfrutta l'interfaccia messa dal client per fargli avere sempre la lista aggiornata degli utenti. Viene sfruttato il meccanismo delle CallBack.
- **regCallback** : effettua la registrazione alle Callback, di fatto viene aggiunto uno nuovo utente, definito da username e la sua interfaccia (EventNotificationInterface), all'ArrayList che li memorizza.
- **unregCallback** : opposto al precedente.
- **notifyChatConnection** : scorre l'ArrayList delle Callback dei client definito precedentemente e segnala la creazione di una nuova chat (a seguito della crezione di un nuovo progetto, o all'aggiunta di un nuovo membro a quest'ultimo), passando al client il suo multicast IP e la porta.
- **notifyChatConnection** : opposto al precedente (in caso di rimozione di un progetto).

3.4 ProjectsManager

Questa entità implementa tutte le operazioni che riguardano i progetti e la loro gestione.

3.4.1 Dati

- **private HashMap<String,Project> projects**
  Mappa il nome del progetto all'oggetto che lo rappresenta, con tutte le sue informazioni all'interno.

- **private final ArrayList<Triplet<String,Integer,MulticastSocket> > mSockets**
  Lista contenente gli indirizzi IP multicast, le porte, e i MulticastSocket dei vari progetti, organizzati sotto forma di tripla. La MulticastSocket in particolare serve al server per comunicare sulla chat messaggi di spostamento delle cards.

3.4.2 Metodi

- **ProjectsManager** (costruttore) : configura de directory e i filepath dei file JSON di persistenza. Se esistono dei file con i dati già esistenti, allora carica in memoria gli oggetti dal JSON, deserializzandoli (in particolare i progetti) e configura i socket delle chat. Dopo di che viene configurato l'AddressGenerator, un tool che serve per l'assegnamento di indirizzi multicast unici ai progetti, anche il suo stato viene mantenuto in un file di permanenza.
- **getProjects** : restituisce la struttura dati contenente i progetti.
- **listProjects** : itera sui progetti, restituendo una coppia <risultato operazione, lista con i nomi dei progetti a cui partecipa l'utente >.
- **checkProject** : verifica la presenza di un dato progetto.
- **createProject** : controlla per prima cosa che il nome del progetto che si vuole creare non sia già presente nel sistema, dopo di che viene creato un nuovo progetto, gli viene assegnato l'indirizzo multicast per la chat, e i cambiamenti vengono scritti sui file di permanenza. Successivamente all'utente che crea il progetto viene notificata, tramite Callback, la presenza di una nuova chat da tenere "sotto controllo" sul client. Alla fine il server configura il suo MulticastSocket per comunicare movimenti delle card ai client. Come risultato viene restituito il codice rappresentante l'esito dell'operazione.
- **addMember** : metodo che inserisce un nuovo utente nella lista dei membri di un dato progetto, controllando prima l'esistenza dell'utente e del progetto in questione. Dopo di che viene aggiornato il file di persistenza dei progetti e viene restituito il codice dell'esito.
- **showMembers** : metodo che restituisce una lista con i nomi dei membri di un determinato progetto, se tale esiste e se l'utente che ne fa richiesta è anch'esso membro del suddetto progetto. Viene restituita una coppia <codice risultato, lista con i nomi dei membri oppure null >.
- **showCards** : controlla l'esistenza del progetto di cui si vuole la lista delle cards, controlla che l'utente sia membro del progetto e in caso di successo restituisce una coppia con <codice risultato,lista di coppie <nome card, stato> >.
- **showCard** : fa gli stessi controlli del metodo precedente, con in più il controllo sull'esistenza della card che si vuole vedere, alla fine restituisce <codice risultato, tripla con le informazioni della card>
- **addCard** : dopo i dovuti controlli inserisce la card nella lista delle card del progetto, restituendo il codice dell'esito. Si appoggia sul metodo createCard implementato direttamente all'interno della classe **Project**.
- **moveCard** : sposta una card da una lista all'altra, facendo appoggio sul metodo moveCard implementato all'interno della classe **Project**, che esegue il vero lavoro. Successivamente viene inviato un messaggio sulla chat del progetto, segnalante lo spostamento eseguito, in caso di succeso. Restituisce il codice di ritorno.
- **getCardHistory** : restituisce la lista dei movimenti di una data card (memorizzata all'interno della classe **Card**) del progetto, se tali esistono, invocando il metodo omonimo della classe **Project**. La struttura dati di ritorno è così composta : <codice ritorno, <lista degli stati che ha attraversato la card> >.

- **getChatInfo** : restituisce la coppia <codice ritorno, indirizzo IP della chat del progetto in questione>, se l'utente che lo invoca ne è membro, ovviamente.
- **cancelProject** : se l'utente che lo invoca è membro del progetto, e quest'ultimo è stato completato (cioè tutte le sue card sono in stato "DONE"), allora il progetto viene rimosso dalla lista dei progetti, se questa diventa vuota, allora si resetta anche AddressGenerator, che può ripartire da capo, in quanto non c'è il rischio di assegnare un indirizzo doppio a due o più progetti. Successivamente vengono aggiornati i file di persistenza e viene rimossa la directory del progetto. Dopodiche vengono effettuate le Callback ai client per avvisarli che la chat relativa al dato progetto non è più attiva, con la conseguente rimozione della tripla che memorizzava le informazioni, insieme alla MulticastSocket della chat. Infine viene restituito il codice con l'esito dell'operazione.
- **updateProjectsJSON** : metodo che scrive il file di permanenza dei progetti (serializza la HashMap dei progetti).
- **updateAddressGenJSON** : metodo che scrive il file di permanenza del generatore degli indirizzi multicast (serializza AddressGenerator).

3.5 Classi Project e Card

Classe **Project**
Questa classe mantiene i dati riguardanti il nome, i membri, le card, e lo stato delle card di un progetto:

- **private final HashSet<String> members** : insieme con i nomi dei membri
- **private transient HashSet<Card> cards** : insieme delle card
- **private final HashSet<String> toDo**
  private final HashSet<String> inProgress
  private final HashSet<String> toBeRevised
  private final HashSet<String> done :
  insieme contenenti i nomi delle card, per tenere traccia del loro stato a runtime, senza dover iterare tutto l'insieme delle card.

Inoltre in questa classe vengono implementati alcuni metodi su cui fa appoggio il ProjectManager, come detto precedentemente, tra cui : showCards per visualizzare tutte le card del progetto, getCardInfo per visualizzare le informazioni di una determinata card, getCardHistory per vedere lo storico degli spostamenti di una card, createCard per l'inserzione di una nuova card nel progetto, con anche la parte sulla persistenza delle sue informazioni, moveCard per lo spostamento delle card.

Classe **Card**
In questa classe vengono mantenute le informazioni riguardanti una card, ovvero il suo nome, la sua descrizione, lo stato attuale, e la lista dei movimenti.

4 WORTH Client

4.1 Dati

- **Indirizzo e porte (TCP e RMI) del server**
- **private List<User> users** : lista degli utenti del servizio, con il loro stato. Viene costantemente aggiornata con le Callback
- **private String username** : username dell'utente
- **private final ArrayList<Chat> chats** : lista contenente le chat dei vari progetti. La classe Chat verrà trattata successivamente
- **ThreadPool chatExec** : per l'esecuzione dei thread che staranno in ascolto sulle varie MulticastSocket delle chat e memorizzeranno i messaggi.
- **Socket e canali di comunicazione TCP con il server**.

4.2 Funzionamento

Il client, dopo aver configurato la connessione RMI, ed aver esposto l'interfaccia per le Callback degli aggiornamenti, cicla leggendo stringhe dalla linea di comando, interpretandole ed eseguendo le varie operazioni a seconda del comando ricevuto, quasi sempre si tratta di inviare tramite **BufferedWriter** il comando al server, appendendo lo username dell'utente, e mettersi in ricezione sull'**ObjectInputStream** in attesa della risposta del server. Se viene rilevata una delle eccezioni che riguardano la connessione TCP con il server, il client termina avvisando l'utente.
Il client dunque, a differenza del server, è single-thread, con però i thread di supporto per le chat.

4.3 Comandi

In figura i comandi per l'interazione :

```
REGISTER 'username' 'password' -> Command used to register a new user.
LOGIN 'username' 'password' -> Command used to login.
LOGOUT -> Command used to logout.
LISTUSERS -> Command used to list all the users
LISTONLINEUSERS -> Command used to list online users.
LISTPROJECTS -> Command used to list the projects where user is member
CREATEPROJECT 'project_name' -> Command used to create a new project
ADDMEMBER 'project_name' 'new_member_nickname' -> Command used to add a user as a member to the project
SHOWMEMBERS 'project_name' -> Command used to list the members of the project
SHOWCARDS 'project_name' -> Command used to list the cards of the project
SHOWCARD 'project_name' 'card_name' -> Command used to show information about a specific card in the project
ADDCARD 'project_name' 'card_name' -> Command used to add a new card to the project
MOVECARD 'project_name' 'card_name' 'source_list' 'destination_list' -> Command used to move a card from a list to another (POSSIBLE
 LISTS: TODO, INPROGRESS, TOBEREVISED, DONE)
GETCARDHISTORY 'project_name' 'card_name' -> Command used to get the history of the movements of the specified card in the project
READCHAT 'project_name' -> Command used to read the messages sent in the project's chat
SENDCHATMSG 'project_name' -> Command used to send a message in the project's chat
CANCELPROJECT 'project_name' -> Command used to cancel a project.
```

5 Altre classi e interfacce

- **User** : classe che memorizza lo username e stato dell'utente.
- **Chat** : classe che serve per la gestione della chat di un progetto lato client, i thread del pool che leggono le chat memorizzano i messaggi qua dentro, in **private ArrayList<String> messages**.
- **CallbacksInfo** : classe che mantiene l'interfaccia del client per le Callbacks e il suo username per l'identificazione.
- **AddressGenerator** : classe che implementa il generatore degli indirizzi IP multicast per le chat.
- **ChatThread** : thread che sta costantemente in ricezione dei datagrammi UDP per la chat lato client, e li memorizza.
- **EventNotificationInterface** : interfaccia che espone il client per ricevere le callback dal server.
- **EventNotificationImp** : classe che implementa la suddetta interfaccia.
- **Pair** : generica classe coppia, caratterizzata da due elementi, di tipo Raw Type, deciso al momento della creazione di un oggeto, serve per ritornare diversi valori di ritorno dal server al client.
- **Triplet** : generica classe tripla, estende la coppia, ha lo stesso scopo.
- **RegInterface** : interfaccia che espone il server al client per la registrazione degli utenti, per l'iscrizione/disiscrizione alle/dalle callback dei client.
- **ServerUnreachableException** : RuntimeException lanciata nel caso in cui un metodo (es.Login) del client si accorga che è caduta la connessione, e deve ritrasmettere l'eccezione al Main.