# PROGETTO WORTH

Massagli Lorenzo

2020-12-26

# Indice

# 1 Scelte di progetto

Il progetto è diviso in due classi main principali: Client e Server.
Il client si connette tramite TCP al server per poter inviare i vari comandi e usa il meccanismo RMI per gli eventi di registrazione e callbacks di aggiornamento.
Il server gestisce le richieste da parte dei client.
Le callbacks si dividono in 2 tipi principali:

- **Aggiornamento lista utenti**: Sono callbacks che vanno ad aggiornare la lista (salvata dai client) degli utenti che sono registrati e il loro status (online/offline);
- **Aggiornamento delle informazioni delle connessioni Multicasting**: Vengono aggiornate le informazioni usate dai client per poter inviare e ricevere i messaggi (Multicasting).

I comandi effettuati dal client vengono sempre inviati tramite TCP al server, il quale verificherà la validità del comando e andrà a svolgere le operazioni necessarie. Gli unici due comandi che non rispettano quanto scritto sono: Register(), listUsers() e listOnlineUsers(). Tra cui gli ultimi due andranno a prelevare le informazioni direttamente dalle liste salvate dai client.
La gestione dei thread è effettuata tramite Multiplexing dei canali mediante NIO e nel caso dei metodi RMI tramite la keyword "Synchronized". Le chat dei progetti sono implementate tramite comunicazioni Multicast dai client. Le informazioni di Multicast, come descritto sopra, vengono aggiornate tramite Callbacks da parte del server. L'interazione client-server è effettuata tramite interfaccia a linea di comando.

## 1.1 Persistenza

La persistenza del server è assicurata tramite dei file JSON che salvano le varie informazioni durante l'esecuzione del server.

- **Users.json**: File che salva le informazioni degli utenti;
- **Projects.json**: File che salva le informazioni dei progetti;
- **MipGenerator.json**: File che salva le informazioni riguardante il generatore di indirizzi Multicast.

Inoltre per ogni progetto viene creata una cartella che conterrà i vari file JSON relativi alle card di quest'ultimo. Le password degli utenti vengono criptate tramite una funzione di hash crittograficamente sicura che prende in input un "salt" (generato random) e la "password" e restituisce una "securePassword".

$$h(salt, password) = securePassword.$$

Vengono poi salvate nel file degli utenti la **securePassword** e il **salt**. Il server, quando sarà richiesto il login da parte di un utente, calcolerà $insertedPass = h(salt, password)$.
Se $insertedPass$ è uguale a $securePassword$ salvata, allora l'utente viene identificato.

# 2 Threads e strutture dati

Nel progetto vengono attivati solo e soltanto i thread relativi al client e al server. La gestione della concorrenza da parte del server viene effettuata tramite Multiplexing dei canali mediante NIO.
Sui metodi invece RMI che offre il server, viene gestita la concorrenza tramite la keyword Synchronized che permette l'accesso a quei metodi (e di conseguenza alle strutture dati), in mutua esclusione.
Le strutture dati utilizzate sono principalmente delle ArrayList, sulle quali viene garantito il controllo della concorrenza come descritto sopra.

# 3 Classi

Le classi che contengono il main sono: clientMain e serverMain.
Le classi principali oltre le classi clientMain e serverMain sono le seguenti:

- **User**: Classe che contiene le informazioni relative agli utenti;
- **Project**: Classe che contiene le informazioni relative ai progetti;
- **Card**: Classe che contiene le informazioni relative alle card dei progetti.

Come classi di supporto nell'implementazione del progetto, sono state create ed utilizzate le seguenti classi:

- **CallBackInfo**: Classe utilizzata per poter associare il nome dell'utente alla ClientInterface per la callback;
- **chatINFO**: Classe utilizzata per poter mandare al client l'indirizzo IP del multicast e il codice associato alla richiesta;
- **multicastINFO**: Classe utilizzata per poter salvare le informazioni delle connessioni multicast (indirizzo IP, porta);
- **MulticastConnectionInfo**: Classe utilizzata per poter salvare le informazioni delle connessioni multicast (MulticastSocket, indirizzo IP, porta), estende la classe "multicastINFO";
- **MulticastIPGenerator**: Classe utilizzata per generare indirizzi IP multicast;
- **NicknameStatusPair**: Classe utilizzata per associare il nome di un utente al suo status (online,offline);
- **Result**: Classe generale utilizzata per poter mandare al client il risultato di un comando;
- **LoginResult**: Classe utilizzata per poter mandare al client il risultato del comando Login, estende la classe "Result";
- **PasswordUtils**: Classe che contiene i metodi usati per criptare e decriptare la password.

# 4 Istruzioni di compilazione

Il codice non richiede argomenti iniziali.

## 4.1 Librerie esterne

Il progetto utilizza solo una libreria esterna: Jackson.
Per installarla:

- Aggiungere i file JAR al classpath;
- Aggiungere le dipendenze Maven.

### 4.1.1 Aggiungere i file JAR al classpath

Scegliere la versione e scaricare le JSON API.
La lista delle releases può essere trovata a questo link:
`https://github.com/FasterXML/jackson-core/releases`.

### 4.1.2 Aggiungere le dipendenze Maven

Aggiungere Jackson come dipendenza al POM file del progetto.

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.9.6</version>
</dependency>

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>2.9.6</version>
</dependency>

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.6</version>
</dependency>
```

Figure 1: dipendenza Maven Jackson

## 4.2 Comandi

La sintassi dei comandi utilizzabili è visualizzabile utilizzando il comando "help".

```
help
---------------- Commands syntax ----------------
ATTENTION: ALL THE COMMANDS ARE NOT CASE SENSITIVE!
register 'nickUtente' 'password' -> Command used to register a new user to the system
login 'nickUtente' 'password' -> Command used to login in the system with a specific user
logout 'nickUtente' -> Command used to logout from a specific user
listUsers -> Command used to list the users registered in the system
listOnlineUsers -> Command used to list the users that are online in the system
listProjects -> Command used to list the projects that the user is member
createProject 'projectName' -> Command used to create a new project
addMember 'projectName' 'nickUtente' -> Command used to add a user as a member in the project
showMembers 'projectName' -> Command used to show the members of the project
showCards 'projectName' -> Command used to show the cards of the project
showCard 'projectName' -> Command used to show the info of a specific card in the project
addCard 'projectName' 'cardName' -> Command used to add a new card to the project
moveCard 'projectName' 'cardName' 'StartingList' 'DestinationList' -> Command used to move a card from a list to another (POSSIBLE LISTS: TODO, INPROGRESS; TOBEREVISED, DONE)
getCardHistory 'projectName' 'cardName' -> Command used to get the history of the movements of a card in the project
readChat 'projectName' -> Command used to read the messages sent in the project's chat
sendchatmsg 'projectName' -> Command used to send a message in the project's chat
cancelProject 'projectName' -> Command used to cancel a project in the system
---------------- Commands syntax ----------------
```

Figure 2: Comando Help