Laboratorio di Reti di Calcolatori

a.a 2020/2021

Antonio Guzzi

581947

Corso B

Introduzione:

Dentro la cartella del progetto sono presenti: la cartella *“src”* contenente i file .java all’interno dei quali sono implementate le varie classi e interfacce utilizzate nel corso del progetto, un file *“help.txt”* utilizzato per la visualizzazione dei comandi disponibili e per le informazioni di utilizzo del server WORTH e una cartella di Backup chiamata *“recoveryDir”*, quest’ultima contenente una serie di file .json per mantenere la persistenza dello stato del server una volta avvenuta la sua terminazione. La *“recoveryDir”* viene gestita nel modo seguente:

- Ogni qual volta un utente si registra al servizio WORTH viene aggiornato il file *“registerdUsers.json”* contenente le informazioni di tutti gli utenti registrati.
- Ogni qual volta un progetto viene creato, viene generata una cartella che riporta il suo nome, quest’ultima conterrà al suo interno i vari file .json relativi alle card di un progetto e un file *“projectMembers.json”* nel quale saranno presenti gli username dei membri del progetto.

I file contenuti all’interno della cartella di Backup vengono aggiornati durante tutto il corso dell’esecuzione del programma per garantire un corretto ripristino, quest’ultimo realizzato tramite l’utilizzo della classe *ObjectMapper*.

Server:

Il server è realizzato mediante l’utilizzo di due classi:

- **Server:** è la classe che implementa gli handlers per le richieste del client, contiene sia alcuni metodi utilizzati in appoggio ai gestori delle richieste sia le strutture dati utilizzate dal server per la gestione delle informazioni.
- **MainClassServer:** contiene la funzione main del server, al suo interno viene istanziato un oggetto di tipo *“Server”* il quale permette l’avvio del server tramite un metodo apposito.

Nel momento in cui il server viene avviato, se la cartella di Backup non esiste viene creata, alternativamente si procede con la sua lettura e con il ripristino della sessione precedente salvando le informazioni all’interno di opportune strutture dati tra cui

`List<User> users` e `ArrayList<Project> projects`

la prima è una struttura di tipo *synchronizedList* che contiene le informazioni riguardanti gli utenti registrati (rappresentati tramite la classe “*User*”), la seconda è la struttura di tipo *ArrayList<Project>* adibita al mantenimento delle informazioni riguardanti i progetti (anch’essi rappresentati tramite l’opportuna classe “*Project*”). La scelta di utilizzare una struttura dati di tipo *synchronizedList* per mantenere le informazioni degli utenti in modo consistente è giustificata dal fatto che, sia condivisa tra due istanze diverse.

Utilizzo di RMI:

Tramite il metodo `registerUser()` il server esporta la variabile di istanza di tipo *EventManager* sulla porta 5000 per ottenere la stub da registrare sotto il nome di “EVENT_MANAGER”.

Sistema di registrazione:

Una volta esportato l'oggetto *EventManager* e registrato il suo riferimento, il client ha la possibilità di utilizzare i metodi disponibili al suo interno, il metodo

```
register (String nickName, String password)
```

è uno di questi, il suo compito è quello di aggiornare il file *"registerdUsers.json"* e creare un nuovo oggetto di tipo *User* da aggiungere alla struttura dati del server passatagli come riferimento all'interno del costruttore. Nel caso in cui lo username, passato come riferimento al metodo, risulti essere già presente all'interno della struttura dati mantenuta dal server, il metodo ritornerà un booleano *false*, quest'ultimo verrà catturato dal client una volta eseguita la chiamata che stamperà in seguito un messaggio di errore.
Una volta terminata l'aggiunta del nuovo utente nella *List<User>*, il metodo invierà una notifica a tutti i client che risultano registrati al sistema di Callback, tale notifica permetterà l'aggiornamento della struttura dati utilizzata dal client anche per mantenere le informazioni sullo stato (Online/Offline) degli utenti registrati al servizio WORTH.

Sistema di notifiche e Callback:

```
registerForCallback (NotifyEventInterface ClientInterface)
```

&

```
unRegisterForCallback (NotifyEventInterface ClientInterface)
```

sono alcuni dei restanti metodi contenuti nella classe *EventManager*, quest'ultimi permettono rispettivamente di registrarsi al servizio di Callback e di eliminare tale registrazione. Per poter fornire un servizio di notifiche asincrone nel caso di eventi come la registrazione, il login e il logout, la classe *EventManager* mantiene una struttura dati di tipo *List<NotifyEventInterface>*, quest'ultima contiene le *NotifyEventInterface* dei client che hanno effettuato l'operazione di registrazione al sistema di notifiche. Nel momento in cui si verifica uno degli eventi descritti in precedenza, il server invoca il metodo

```
update(String map)
```

il quale scorre la lista di *NotifyEventInterface* e consente l'esecuzione del metodo di aggiornamento proposto dall'interfaccia del client, l'aggiornamento avviene sulla base della lista di *Users* resa nota dal server sotto forma di stringa, tale scelta implementativa è stata utilizzata allo scopo di non rendere note al client informazioni sensibili ad esso non necessarie.

Utilizzo di TCP:

il server, una volta avviato, si mette in ascolto sulla porta 5001 in attesa di richieste di connessione da parte dei client. Per l'esecuzione delle richieste il server effettua il multiplexing dei canali tramite l'utilizzo di Java NIO e, in particolar modo, dei selettori. L'utilizzo dei selettori come tecnica implementativa del server permette la realizzazione di quest'ultimo come un unico thread che gestisce più connessione TCP con client diversi, così facendo è stato possibile ridurre l'overhead dovuto al thread switching e l'uso di risorse per thread diversi.
Nel momento in cui il server riceve un comando da un client, controlla di che tipo di operazione si tratti per poter eseguire il metodo corretto. All'interno dei metodi del server,

sono implementati una serie di controlli che permettono di inviare al client eventuali messaggi di errore generati durante l'esecuzione.
Si noti come nella fase di login vengano settati, tramite l'opportuno metodo della classe *User*, l'indirizzo IP e la porta del client richiedente un servizio a WORTH; questa scelta implementativa permette di rendere più comodo il controllo dei diritti del client richiedente in determinati tipi di metodi inerenti la manipolazione di un progetto.

Utilizzo di UDP Multicast:

All'interno della classe *Project,* oltre alle quattro liste di tipo *ArrayList<Card>,* le quali permettono di gestire le carte e i loro spostamenti e oltre alla lista di tipo *ArrayList<String>,* utile per mantenere gli username degli utenti membri del progetto, vengono conservati anche l'indirizzo IP di multicast e la porta relativi alla chat associata al progetto.
Per la gestione delle notifiche inviate sulla chat del progetto, la classe *Project* implementa il metodo

```
sendMessage(String msg)
```

il quale utilizza l'indirizzo IP di multicast e la porta per inviare messaggi riguardanti eventi quali lo spostamento di una card, l'aggiunta di una card e l'aggiunta di un membro al progetto. l'indirizzo IP di multicast è compreso nel range 239.0.0.0 – 239.255.255.255 e viene generato al momento della creazione del progetto tramite un metodo apposito implementato all'interno del server. Per la generazione di indirizzi IP multicast il server implementa il riuso di quest'ultimi attraverso un *ArrayList<String>,* questa struttura dati viene riempita con gli indirizzi IP dei progetti cancellati che possono essere riutilizzati per progetti nuovi.

Client:

Il client è realizzato, così come il server descritto in precedenza, mediante l'utilizzo di due classi:

- **Client:** è la classe che implementa i metodi per effettuare le richieste al server WORTH, al suo interno mantiene due strutture dati:

  `HashMap<String, String> map` e `HashMap<String, Chat> chats`

  una per il mantenimento delle informazioni degli utenti e del loro stato, l'altra per la gestione dei progetti e delle chat ad essi associate.
- **MainClassClient:** contiene la funzione main del client, al suo interno viene istanziato un oggetto di tipo *"Client"* il quale permette l'avvio del client tramite un metodo apposito.

Interazione con il Server WORTH:

nel momento in cui il client viene avviato, eseguirà una lookup del nome "EVENT_MANAGER" sul registro delle stub rese visibili dal server, per poi recuperare l'oggetto remoto esportato da quest'ultimo. In seguito al recupero dell'oggetto remoto il client entrerà in un primo ciclo iniziale all'interno del quale sarà possibile, da parte dell'utente, eseguire solo alcune operazioni tra cui:

- **register:** all'interno del quale avviene l'invocazione del metodo dell'oggetto remoto recuperato nella fase iniziale.
- **login:** all'interno del quale il client esporta un'istanza della classe *NotifyEvent* utile per ricevere le notifiche da parte del server. Nel metodo di login, inoltre, il client si registra al sistema di notifiche implementate tramite Callback grazie al metodo

  `turnUpNotifications (EventManagerInterface remoteEventManager, NotifyEventInterface stub)`
- **quit:** tramite il quale si permette la terminazione del client. Nel caso in cui questo comando venga lanciato durante il primo ciclo iniziale implementa una semplice return, alternativamente, all'interno del comando di quit viene chiamato il metodo per effettuare il logout e il client viene terminato.
- **help:** permette di visualizzare la lista di comandi disponibili per interagire con il server WORTH. All'interno di questo metodo si esegue la lettura di un file .txt tramite l'utilizzo della classe *BufferedReader.*

Una volta completata correttamente la fase di login, il client entra nel secondo ciclo iniziale, all'interno del quale gli sarà possibile eseguire tutti i comandi per interagire con il server WORTH.
Si noti come all'interno dell'implementazione del comando di logout il client esegua sia l'*unexportObject* dell'istanza di *NotifyEvent* esportata in precedenza nella fase di login, sia la chiusura della connessione TCP.

Realizzazione della Chat:

all'utente è resa possibile l'opzione di partecipazione ad una chat di un progetto tramite l'utilizzo del comando join_chat, una volta che il server riceve tale comando sfrutta la connessione TCP inizializzata in precedenza per mandare al client le informazioni necessarie per unirsi alla chat (indirizzo IP multicast e porta associati al progetto). La chat viene modellata attraverso la classe *Chat* che implementa una serie di metodi utilizzati dal client per l'invio dei messaggi e la loro ricezione.
Il costruttore della classe *Chat,* che prende come argomenti un indirizzo IP e una porta, permette di istanziare una *MulticastSocket* e un indirizzo di multicast, i quali verranno utilizzati nella fase di ricezione e invio dei messaggi della chat.
Per la ricezione dei messaggi viene utilizzata la struttura dati

`ArrayList<String> unreadMessages`

che viene costantemente aggiornata all'interno di un thread fino a quando il progetto non verrà chiuso e quindi la chat verrà eliminata; tale evento è segnalato dal server mediante il messaggio "Server WORTH: close". Lo stato della chat è reso noto da una variabile di tipo *AtomicBoolean* che assume il valore *false* solo nel momento in cui la chat viene eliminata, tale variabile viene controllata ciclicamente all'interno del thread per verificare lo stato della chat e del progetto ed evitare inconsistenze.
Per l'invio dei messaggi la chat implementa il metodo

`sendMsg(String msg)`

all'interno del quale si prepara e si invia un datagramma sull'indirizzo di multicast della chat.

Classi utilizzate all’interno del progetto:

Oltre alle classi che modellano il client e il server, all’interno del progetto vengono utilizzate anche alcune classi per rappresentare al meglio alcuni aspetti dell’interazione con il server WORTH, tra cui:

- **User:** è la classe che modella un utente del server WORTH, al suo interno si mantengono lo username dell’utente, la password a lui associata, il suo stato, l’indirizzo IP e la porta del processo client dal quale l’utente esegue il login. Alcuni campi utilizzati all’interno di questa classe non necessitano di essere utilizzati nella fasi di serializzazione e deserializzazione, per questa ragione sono marcati con l’etichetta *“@JsonIgnore”*. L’indirizzo IP e la porta associati all’utente vengono utilizzati nella fase di logout (per riconoscere l’utente che necessita di essere scollegato) e nella fase di verifica delle credenziali nel caso di richieste che vadano a modificare un progetto.

```java
/**
 * @param nickName: nome dell'utente
 * @param psw: passworod associata all'utente
 * @param state: stringa che esprime lo stato Online/Offline dell'utente
 * @throws IllegalArgumentException se il nome utente, la password o lo stato sono nulli
 */
public User(String nickName, String psw, String state) throws IllegalArgumentException{
    if(nickName == null) throw new IllegalArgumentException("nickName null");
    if(psw == null) throw new IllegalArgumentException("psw null");
    if(state == null) throw new IllegalArgumentException("state null");
    this.nickName = nickName;
    this.psw = psw;
    this.state = state;
    this.ip = null;
}
```

- **Project:** è la classe che modella un progetto conservato nel Data Base del server WORTH, al suo interno vengono mantenuti il nome unico del progetto, le liste necessarie per lo spostamento delle carte, una lista contenente gli username dei membri del progetto e l’indirizzo IP multicast e la porta associati alla chat del progetto. In questa classe viene anche istanziato un oggetto di tipo *DatagramSocket* utilizzata nell’operazione di invio dei messaggi.

```java
/**
 * @param projectName: nome da assegnare al progetto
 * @throws IllegalArgumentException se il nome del progetto è nullo
 */
public Project (String projectName) throws IllegalArgumentException{
    if(projectName == null) throw new IllegalArgumentException("projectName null");
    this.projectName = projectName;
    this.TODO = new ArrayList<Card>();
    this.DONE = new ArrayList<Card>();
    this.INPROGRESS = new ArrayList<Card>();
    this.TOBEREVISED = new ArrayList<Card>();
    this.pojectUsers = new ArrayList<String>();

    try{
        this.datagramSocket = new DatagramSocket();
    }catch(IOException e){e.printStackTrace();}
}
```

- **Card:** è la classe che modella le card presenti all'interno di un progetto, al suo interno vengono mantenuti il nome (unico all'interno del progetto), la descrizione e lo storico della card. Questa classe implementa una serie di metodi per rendere più comode le funzioni di gestione delle card implementate dalla classe *Project.*

```java
/**
 * @param name: nome della carta
 * @param description: descrizione della carta
 * @throws IllegalArgumentException se uno dei due paramentri risulta essere uguale a null
 */
public Card(String name, String description) throws IllegalArgumentException {

    if(name == null) throw new IllegalArgumentException("nome null");
    if( description == null) throw new IllegalArgumentException("descrizione null");

    this.cardName = name;
    this.description = description;
    this.history = new ArrayList<String>();
}
```

- **Chat:** è la classe che modella una chat associata ad un progetto, al suo interno mantiene lo username mittente del messaggio, un booleano che indica lo stato della chat e una struttura dati di tipo *ArrayList<String>* per mantenere i messaggi non ancora letti dall'utente. All'interno di questa classe viene implementato un thread per la ricezione dei messaggi su un indirizzo IP di multicast e un metodo per l'invio dei messaggi sulla chat.

```java
/**
 * @param nickName: nickName del partecipante alla chat
 * @param ip: indirizzo IP del partecipante alla chat
 * @param port: porta utilizzata dalla chat per la comunicazione
 * @throws IllegalArgumentException se il nickName dell'utente o il suo indizzo ip sono nulli
 */
public Chat(String nickName, String ip, int port) throws IllegalArgumentException{

    if(nickName == null) throw new IllegalArgumentException("nickName null");
    if(ip == null) throw new IllegalArgumentException("ip null");

    this.nickName = nickName;
    this.ip = ip;
    this.port = port;
    //come la lista dei messaggi non letti, anche la variabile che permette di capire lo stato della chat è modificabile solo tramite
    //operazioni atomiche, questo per via del numero di utenti che possono accedere contemporaneamente alla chat
    this.listening = new AtomicBoolean();
    this.listening.set(false);
    this.unreadMessages = new ArrayList<String>();

    try {
        this.multicastAddress = InetAddress.getByName(this.ip);
        this.group = new MulticastSocket(this.port);
        this.datagramSocket = new DatagramSocket();
        this.group.joinGroup(multicastAddress);
    } catch (IOException e) {e.printStackTrace();}
}
```

- **NotifyEvent:** è la classe che implementa l'interfaccia *NotifyEventInterface*, questa classe contiene i metodi con i quali il client viene notificato allo scaturirsi di eventi quali la registrazione di un nuovo utente, il login e il logout di un utente registrato. I metodi di notifica lavorano sulla struttura dati del client (contenente i nomi degli utenti e il loro stato) passata per riferimento al costruttore.

```java
/**
 * @param map: struttura dati del client che necessita di essere aggiornata
 * @throws RemoteException
 */
public NotifyEvent(HashMap<String, String> map) throws RemoteException{
    super();
    if(map == null) throw new IllegalArgumentException("struttura dati 'map' null");
    this.map = map;
}
```

- **EventManager:** è la classe che implementa l'interfaccia *EventManagerInterface*, questa classe viene utilizzata dal client per eseguire i metodi dell'oggetto remoto messo a disposizione dal server sul registro caricato sulla porta 5000 utilizzata da RMI. All'interno di questa classe sono presenti anche i metodi di registrazione e eliminazione della registrazione al servizio di notifiche.

```java
/**
 * @param users: lista degli utenti registrati
 * @throws RemoteException
 */
public EventManager(List<User> users) throws RemoteException{
    if(users == null) throw new IllegalArgumentException("struttura dati 'users' null");
    this.users = users;
    this.clients = new ArrayList<NotifyEventInterface>();
}
```

Esecuzione da terminale:

Per la compilazione è necessario trovarsi all'interno della cartella del progetto

compilazione del progetto: javac -cp ./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar: ./src/progetto_2020_2021/*.java -d bin

esecuzione del server: java -cp ./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar:./bin: progetto_2020_2021.MainClassServer

esecuzione del client: java -cp ./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar:./bin: progetto_2020_2021.MainClassClient

Comandi utilizzabili dall'utente:

- *register* [nome dell'utente] [password]
- *login* [nome dell'utente] [password]
- *quit*
- *help*
- *logout*
- *list_users*
- *list_online_users*
- *list_projects*
- *create_project* [nome del progetto]
- *add_member* [nome del progetto] [nome dell'utente]
- *show_members* [nome del progetto]
- *show_cards* [nome del progetto]
- *show_card* [nome del progetto] [nome della carta]
- *add_card* [nome del progetto] [nome della carta] [descrizione della carta]
- *move_card* [nome del progetto] [nome della carta] [sorgente] [destinazione]
- *get_card_history* [nome del progetto] [nome della carta]
- *join_chat* [nome del progetto]
- *read_chat* [nome del progetto]
- *send_chat_msg* [nome del progetto] [messaggio]
- *cancel_project* [nome del progetto]