Documentazione progetto “WORTH”

Fabio Guastapaglia, 588246

Indice

1. Architettura del sistema

Il sistema è composto da due eseguibili: **client** e **server.**
Il **client** appena avviato effettua una connessione al server tramite TCP e si mette in attesa di comandi che arrivato da linea di comando. Digitando *'help'* è possibile mostrare a video la lista di tutti i comandi disponibili, ma gli unici che verranno accettati inizialmente dal server sono login o register. Il comando register effettua una registrazione al servizio tramite RMI e in caso di successo inoltra il comando di login tramite TCP. Nella procedura di login il client si registra alle notifiche di tipo RMI callback che gli permetteranno di ricevere oggetti di tipo Notification. L'oggetto di tipo Notification contiene una lista degli utenti registrati con associato lo stato attuale che può essere online oppure offline, e una lista dei progetti di cui l'utente fa parte, associato all'indirizzo IP multicast della relativa chat.
All'arrivo dell'oggetto Notification il client aggiorna la sua lista locale di coppie <username, stato>, crea una nuova chat per ogni nuovo progetto (viene avviato un thread per ogni chat) a cui l'utente è stato aggiunto (se quindi è la prima notifica che riceve, creerà le chat per tutti i progetti di cui fa parte), elimina le chat dei progetti che sono stati eliminati.
Il **server,** quando viene avviato, ripristina i dati relativi ad utenti e progetti presenti nella cartella 'storage', dopodiché gestisce le chiamate RMI tramite metodi synchronized e le richieste TCP tramite il multiplexing dei canali mediante NIO. Quando un client effettua il login con successo, la key del suo canale viene salvata nell'oggetto di tipo User corrispondente all'utente per il quale ha richiesto il login. Key e User rimangono associati fino a che non viene effettuato un logout. Nello stesso modo anche alla richiesta di registrazione alla callback RMI l'interfaccia remota del client viene salvata nell'oggetto User corrispondente e viene cancellata nel momento dell'annullamento della registrazione. Le callback RMI vengono effettuate successivamente alla creazione o cancellazione di un progetto, all'aggiunta di un nuovo membro al progetto o al login da parte di un utente.

2. Classi

Sono state implementate diverse classi tra cui

- **Card;**
- **Project;**
- **ProjectSet;**
- **User;**
- **UserSet;**
- **Notification;**
- **Esito;**
- **ChatProcess;**
- **StorageManager;**
- **ClientMain**;
- **ServerMain;**

Oltre a queste abbiamo la classe *Utils,* che contiene diverse funzioni di supporto come, ad esempio, quella per la creazione di un indirizzo IP e l'interfaccia *ServerInterface* che permette la registrazione e la cancellazione per la callback. Sono state inoltre definite delle eccezioni: *CardNotFoundException, IllegalChangeStateException, MultipleLoginException, ProjectNotFoundException, UserAlreadyLoggedException e UserNotFoundException.*

2.1 Card

È la diretta implementazione delle card all'interno dell'applicazione.

È caratterizzata da

- Nome rappresentato da una stringa;
- Descrizione rappresentata attraverso una stringa;
- Stato corrente *(CurrentState*), di tipo *cardStatus* definito per enumerazione: {TODO, INPROGRESS, TOBEREVISED, DONE};
- Cronologia degli stati *(cardHistory),* rappresentato mediante un ArrayList<cardStatus>.

Fornisce i seguenti metodi:

- getName(), setName(*Name*) restituiscono e settano rispettivamente il nome della Card;
- GetDescription(), setDescription(desc) restituiscono e settano rispettivamente la descrizione;
- GetCurrentState(), setCurrentState(*CurrentState*) restituiscono e settano rispettivamente lo stato attuale della Card;
- GetCardHistory(), setCardHistory(*history*) restituiscono e settano rispettivamente la cronologia degli stati della Card;

2.2 Project

È la diretta implementazione dei progetti all'interno dell'applicazione.

È caratterizzato da

- Nome rappresentato da una stringa;
- *cards*, una lista delle card appartenenti al progetto (*List<Card>*);
- Una lista di nomi di card per ogni possibile stato di esse;
- Una lista di membri *(List<String>)* del progetto
- Un indirizzo multicast del progetto IPMulticast rappresentato da una stringa;
- Una variabile final List<Project> PROJECT_CREATED contenente tutti i progetti che sono stati creati, questa servirà per permettere di non creare due progetti con lo stesso indirizzo multicast. Quando viene chiamato il costruttore della classe, questo infatti prima di assegnare l'indirizzo IP creato attraverso la funzione ausiliare randomMulticastipv4() presente nella classe Utils, controlla che questo non sia già stato assegnato ad un altro progetto presente in PROJECT_CREATED. Alla fine dell'esecuzione del costruttore poi il progetto verrà aggiunto a questa lista.

Questa classe fornisce i seguenti metodi:

- GetName() che restituisce il nome del progetto in questione,
- AddMember(*utente*) che aggiunge un utente alla lista dei membri del progetto;
- IsMember(utente) che dato un utente restituisce true se l'utente fa parte dei membri del progetto, false altrimenti;
- GetMembers() che restituisce la lista di tutti i membri del progetto;
- SetMembers(*memb*) che data una lista di stringhe (nomi degli utenti) la setta come membri del progetto;
- GetIPMulticast(), restituisce l'indirizzo IP multicast del progetto;
- AddCard(*card*), aggiunge una card alla lista delle Card del progetto *(cards)* e aggiunge inoltre della card alla lista corrispondente allo stato di quella card, se non è già presente un'altra card con lo stesso nome;

- CreateCard(*name, description*), crea una card a partire da nome e descrizione mediante il costruttore della classe Card e la aggiunge al progetto se non è già presente un'altra card con lo stesso nome;
- GetCard(*name*), restituisce un oggetto di tipo Card corrispondente alla card con quel nome se esiste una card con nome 'name' facente parte di quel progetto;
- ChangeCardState(*name, oldStatus, newStatus*) cambia lo stato della Card 'name' da 'oldStatus' a 'newStatus' se è un cambio legale, altrimenti da un'eccezione;
- GetAllCards() restituisce la lista di tutte le card facenti parte del progetto;
- GetCardHistory(*name*), getCardInformation(*name*) restituiscono rispettivamente cronologia degli stati e informazioni della card 'name' utilizzando i metodi della classe Card;
- GetCardStringList() restituisce una lista di stringhe rappresentate i nomi di tutte le Card facenti parte del progetto;
- IsDone() restituisce true se tutte le Card sono nello stato DONE, false altrimenti. Questo ci permette di sapere se il progetto è stato completato o meno.

2.3 StorageManager

Classe che si occupa di ripristinare dati e aggiornare i file json presenti nella cartella storage, che rappresenterebbe il punto di salvataggio e ripristino dell'applicazione.

Caratterizzata da

- Un ObjectMapper che permetterà di mappare coppie di valori nei file json;
- Una stringa contenente il path della cartella Storage;
- Una stringa contenente il path della cartella contenente i progetti;
- Una stringa contenente il file path del file contenente tutti gli utenti con relativa password registrati sull'applicazione;
- Una stringa contenente '*member.json'* ovvero il file contenente la lista dei membri dei progetti.

All'avvio dell'applicazione questa classe permette il ripristino degli utenti (*restoreUsers()*) e dei progetti (*restoreProjects()*) presenti nella cartella storage, quest'ultima se non è presente all'istanziazione della classe viene creata (vuota). Questa classe contiene anche i seguenti metodi:

- UpdateProjects(*ArrayList<Project> projects*) che permette di aggiornare le sottocartelle di Storage contenenti i progetti solitamente quando ne vengono aggiunti o eliminati salvando i nuovi progetti con le relative Cards nella nuova cartella creata e creando un file json contenente tutti i membri di quel progetto. Ad ogni Card viene associato un file json contenente nome, descrizione, cronologia degli stati e stato attuale della card a cui fa riferimento;
- UpdateUsers(*ArrayList<User>* users) aggiorna il file contenente gli utenti quando solitamente quando un nuovo utente si registra, ogni volta ricrea il file da zero inserendo tutte le coppie utente password (quest'ultima criptata attraverso un metodo della classe Users che richiamerà a sua volta un metodo della classe Utils.

2.4 ProjectSet

Classe contenente la lista degli oggetti Project e ha i metodi per creazione e cancellazione di progetti, restituire tutti i progetti di cui fa parte un utente o restituire un progetto.

È caratterizzato da

- Projects *(ArrayList<Project>)* lista dei progetti;
- Storage *(StorageManager)*

Il costruttore di questa classe inserisce tutti i progetti in 'projects' attraverso il metodo restoreProjects() della classe StorageManager.

Ha inoltre i seguenti metodi

- GetProjectByName(*String name*) che restituisce un oggetto di tipo Project a partire dal nome;
- AddProject(name) che permette la creazione di un progetto con nome 'name' se non esiste già un progetto con quel nome e inoltre lo aggiunge allo storage dell'applicazione mediante il metodo updateProjects dello StorageManager;
- DeleteProject(*String name*) che permette l'eliminazione di un progetto a partire dal nome;
- DeleteProject(*Project project*) elimina il progetto che viene passato come argomento alla funzione;
- CreateCard(*pName, cName, desc*) che permette la creazione di una card con nome 'cName' e descrizione 'desc' nel progetto 'pName';
- GetCards(*pName*) restituisce una lista di Card facenti parte del progetto 'pName';
- IsDone(*pName*) restituisce un booleano che e' uguale a true se il progetto 'pName' e' terminato e false altrimenti. Utilizza il metodo isDone della classe Project;
- GetCardHistory(*pName, cardName*) restituisce la cronologia della Card *cardName* facente parte del progetto *pName* mediante il metodo della classe Project
- GetCardInfo(*pname, cardName*) restituisce le informazioni relative alla card *cardName* facente parte del progetto *pName.* Per informazioni si intende nome, descrizione e stato corrente;
- AddMember(*pName, String user)* aggiunge l'utente *user* alla lista dei membri del progetto *pName;*
- IsMember(pName, user) restituisce un booleano che e' uguale a true se l'utente *user* fa parte del progetto *pName*, false altrimenti;
- ChangeCardState(pName, cName, oldStatus, newStatus) permette di cambiare lo stato di una card mediante il metodo della classe Project;
- GetCardList(*pName*) restituisce la lista delle Card facenti parte del progetto *pName* attraverso il metodo della classe Project;
- ListProject(*user*) restituisce la lista dei nomi di tutti i progetti di cui un utente fa parte;
- UpdateProjects() richiama il metodo updateProjects della classe StorageManager;
- GetChatList(*user*) restituisce la lista dei messaggi per un utente dai relativi progetti di cui fa parte;

2.5 User

Classe che rappresenta un utente ed è caratterizzata da username, password, saltKey (chiave di cifratura della password), stato ed interfaccia remota del client. L'interfaccia remota del client sarà diversa da null se e solo se l'utente è online.

Alla creazione di un nuovo utente viene generata una SaltKey pseudo casuale attraverso la quale verrà poi criptata la password. La funzione di creazione di criptazione della password è descritta nella classe Utils.

La classe User dispone inoltre dei seguenti metodi:

- Notify() utilizza l'interfaccia remota del client per mandare una notifica tramite RMI callback;

- Login(*password)* permette il login dell'utente, avendo salvata la chiave provando a cifrare l'argomento del metodo con la stessa chiave, il risultato deve corrispondere a quello salvato nello storage;
- SetClient(c*lientInterface)* setta l'interfaccia remota del client;
- Notify(notification), utilizza l'interfaccia remota del client per mandare una notifica tramite RMI callback;
- IsOnline(), setOnline(*online*) permettono di sapere se un utente è online e di settare lo stato di un utente;
- GetUsername, setUsername(*username*) restituiscono e settano relativamente l'username di un utente;
- GetSaltKey(), setSaltKey(*saltKey*) restituiscono e settano relativamente la chiave di cifratura dell'utente;
- GetPassword(), setPassword(*psw*) restituiscono e settano relativamente la password di un utente;

2.6 UserSet

Questa classe consente la gestione degli utenti.

È caratterizzato infatti da

- Una lista di oggetti User;
- Storage *StorageManager;*
- Una lista di coppie <key, User> che rappresenta l'associazione tra gli utenti connessi ed il loro canale di connessione.

Metodi:

- Register(*username, password*) permette la registrazione di un utente, restituisce un booleano uguale a true se la registrazione avviene con successo e false altrimenti. Quando un utente si registra risulterà poi anche loggato su quel client senza necessita di eseguire anche quest'ultimo passaggio;
- Login(*username, password, userKey*) permette il login di un utente su un determinato client;
- Logout(*key*);
- SetClient(*username, clientInterface*) setta l'interfaccia remota del client mediante il metodo dell'oggetto User;
- GetByUsername(*username*) metodo synchronized che restituisce un oggetto di tipo User a partire dall'username;
- GetUserList() restituisce una coppia di valori <String, boolean> corrispondente al nome utente e relativo stato;
- NotifyAll(*projects*) notifica tutti gli utenti facenti parte dei progetti facenti parte del *ProjectSet (praticamente tutti);*
- IsLogged(*userKey*) restituisce un booleano uguale a true se l'utente corrispondente a quella chiave è loggato, false altrimenti;
- GetUsers() restituisce una lista di oggetti di tipo User corrispondente a tutti gli utenti registrati;
- GetUsernameByKey(*key*) restituisce una stringa corrispondente all'username a partire dalla chiave associata;
- GetOnlineUsersList() restituisce una lista di stringhe contenente gli username degli utenti attualmente online.

2.7 ChatProcess

Classe utilizzata dal client per creare thread che rimangano in ascolto di messaggi sulla chat, ogni client avrà quindi un thread per ogni progetto di cui l'utente loggato fa parte.

Utilizza una classe di supporto: ***MessageQueue*** che agevola la gestione della coda dei messaggi e permette l'inserzione di messaggi attraverso il metodo *put(String message)* e la "lettura" di messaggi attraverso il metodo *getAndClear().* Una volta che i messaggi vengono letti la coda viene pulita.

La classe ChatProcess è caratterizzata da una coda di messaggi di tipo *MessageQueue,* una porta, un socket multicast e un indirizzo IP del gruppo multicast corrispondente al progetto.

Metodi:

- ReadMsg() restituisce la lista dei messaggi mediante il metodo della MessageQueue;
- SendMessage(*msg*) permette l'invio del messaggio, metodo utilizzato dai client che istanziano l'oggetto;
- SendMsg(*groupAddress, port, message*) metodo statico utilizzato dal server per inviare messaggi, ad esempio, quando viene creata una nuova card in un progetto;
- Receive() permette la ricezione di pacchetti che corrispondono poi ad un messaggio che viene aggiunto tramite il metodo addMsg(*message*) alla coda utilizzando un metodo di MessageQueue;
- Run() serve per la creazione di un thread che resti in attesa sulla chat di un determinato progetto;

2.8 ClientMain

Classe client che legge i comandi e li inoltra al server.

È caratterizzato da:

- Serial version UID;
- Interfaccia del server;
- Username, password e stato di un utente in particolare (quello loggato sul client);
- Una lista degli utenti con lo stato corrispondente;
- Una lista delle chat di progetto;
- L'indirizzo IP del server;
- La dimensione del buffer;
- Le porte relative ai servizi RMI, TCP e CHAT;
- Il socket channel del client;
- Una variabile contenente il comando per la fine dell'esecuzione in questo caso 'exit' e un flag che è true se l'esecuzione del client è stata terminata.

Il costruttore della classe crea un nuovo CallBack client partendo dall'interfaccia del server.

Metodi:

- Login(username, password) permette il login di un utente sul client corrente;
- Register(username, password) permette la registrazione di un nuovo utente;
- Sendcommand(*command*) crea un messaggio corrispondente al comando digitato sul terminale da mandare al server affinché questo possa eseguirlo;
- GetResponse() restituisce un oggetto di tipo Esito e permette di ricevere l'esito di un operazione/comando eseguita dal server;
- Close() termina l'esecuzione del client;
- Start() avvia l'esecuzione del client provando a connettersi al server;

- ListUsers() stampa la lista di tutti gli utenti con relativo stato (synchronized);
- ListOnlineUsers() stampa la lista degli utenti attualmente online (synchronized);
- ReadChat(*chat*) permette di leggere la chat relativa ad un progetto;
- SendChatMsg(*chat, msg*) permette di mandare il messaggio *msg* sulla chat di progetto *chat*;
- Updatechats(*projectChatIPs*) crea chat di eventuali nuovi progetti ed elimina quelle dei progetti non più esistenti;
- Help() stampa a schermo tutti i comandi disponibili;
- ExecuteCommand(*command*) esegue il comando che viene letto da terminale o lo invia al server per esser eseguito;
- Main(*args*);

Il Client quando viene avviato esegue il metodo start() e resta in attesa dei comandi da terminale, quando un comando viene inviato va in esecuzione ExecuteCommand che agisce in questo modo: divide la stringa ricevuta mediante spazi " " in una lista di stringhe di cui la prima è sempre il comando mentre le altre, se presenti, corrispondono agli argomenti del comando. È strutturato mediante uno switch sul comando; alcuni di questi come quelli sulla chat o sugli utenti possono essere eseguiti direttamente da questa classe mentre altri necessitano di esser inviati al server attraverso il metodo *sendCommand,* il quale dopo averli eseguiti ci restituirà l'esito che riceveremo mediante il metodo *getResponse.*

2.9 ServerMain

Classe server che gestisce ed esegue i comandi richiesti dai vari client.

Caratterizzato da

- Serial version UID;
- Un oggetto UserSet e uno ProjectSet che permettono di eseguire le richieste effettuate dal client attraverso i loro metodi;
- Uno StorageManager che permette il persistere dello stato del server;
- Una dimensione del buffer;
- Le porte per i servizi TCP, RMI e CHAT;
- I path relativi allo storage;
- Un ObjectMapper;
- Una variabile contenente il comando per la fine dell'esecuzione in questo caso 'exit'.

Il costruttore della classe crea il server inizializzando un ObjectMapper, uno StorageManager, un UserSet e un ProjectSet e richiamando il costruttore della classe che estende, ovvero di RemoteObjects. La classe implementa inoltre l'interfaccia *ServerInterface* che permette la registrazione e la cancellazione dalla registrazione per le callback.

Metodi:

- RegisterForCallback(*clientInterface, username*) registra un utente *username* sul client *clientInterface* per le RMI callback;
- UnregiregisterForCallback(*username*) annulla la registrazione di un utente per le callback, solitamente quando un client viene chiuso;
- NotifyUsers() notifica tutti gli utenti;
- Register*(username, password*) permette la registrazione di un nuovo utente;
- IterateKeys(Selector *sel*) gestisce le richieste ricevute sul selettore dai vari client;
- Start() avvia l'esecuzione del server dopodiché inizia un ciclo infinito in cui esegue il metodo *iterateKeys(sel);*

- CancelKey(*SelectionKey key*) richiede la cancellazione del canale della chiave *key* e relativo selettore dopo aver chiuso il canale;
- RegisterRead(*sel,* SocketChannel *cChannel*) registra interesse all'operazione di READ sul selettore *sel*, sel rappresenta il selettore utilizzato dal server mentre cChannel e' il socket channel relativo al client. Viene creato un buffer dopodiché il canale del client viene aggiunto al selettore con operazione OP_READ e aggiunge l'array di bytebuffer[lenght, message] come attachment;
- AnswerClient(*sel, key*) scrive il buffer sul canale del client;
- ReadClientMsg(*key*) accetta una nuova connessione creando un socket channel per la comunicazione e recupera l'array di bytebuffer dopodiché lo legge;
- ExecuteCommand(*command, key*) esegue i comandi richiesti dal client;
- Main().

2.10 Notification

Classe utilizzata per le RMI callback e contiene una lista di coppie <utente, stato> e una di coppie <nomeProgetto, IPchat>. Dispone soltanto del costruttore della classe.

2.11 Esito

Classe utilizzata per rispondere alle richieste TCP. Contiene un booleano che rappresenta l'esito dell'operazione richiesta, un messaggio e una lista opzionale di stringhe aggiuntive. Dispone soltanto di tre tipi di costruttori: uno che ha come argomento soltanto l'esito, uno sia esito che messaggio ed uno esito, messaggio e lista di stringhe.

3. Concorrenza

La concorrenza viene gestita così:

- Il client ha un thread per ogni chat che rimane in ascolto di pacchetti UDP sul gruppo multicast; quando arriva un messaggio, viene inserito nella struttura dati MessageQueue. Questa struttura funziona come coda che viene riempita e svuotata in modo sincrono per non creare inconsistenze.
- L'oggetto di tipo UserSet mantiene al suo interno la lista degli utenti registrati che deve esser aggiornata all'invocazione di register da RMI. Si potrebbe creare un conflitto con le richieste TCP che hanno bisogno di accedere alla lista degli utenti come register, notifyAll, getByUsername che sono infatti synchronized.

4. Utilizzo

Il server deve esser mandato in esecuzione prima del client. Una volta in esecuzione stamperà la scritta "Server in attesa di connessioni sulla porta 1919". Successivamente stamperà un messaggio ogni volta che un client si connette o inoltra un comando.

```
Server in attesa di connessioni sulla porta 1919
Server: accettata nuova connessione dal client /127.0.0.1:58948
comando richiesto:login fabio 1234
Server: Login avvenuto con successo inviato al client: /127.0.0.1:58948
```

Il client una volta mandato in esecuzione stamperà “CLIENT CONNESSO” se è riuscito a connettersi al server. Dopodiché stamperà questo messaggio e sarà possibile iniziare ad eseguire dei comandi.

```
CLIENT CONNESSO
------------------------------------------------------------
Digitare help per mostrare la lista dei comandi disponibili
Digitare exit per uscire
>
```

Il client accetta come primi comandi solo quelli di register o login.

Il comando help restituisce una lista di possibili comandi come illustrato:

```
CLIENT CONNESSO
------------------------------------------------------------
Digitare help per mostrare la lista dei comandi disponibili
Digitare exit per uscire
> help
------------------- POSSIBILI COMANDI --------------------
register [username] [password]      registra un nuovo utente
login [username] [password]         effettua login
exit                                  effettua logout
createproject [project name]        crea un nuovo progetto
cancelproject [project name]        cancella un progetto (tutte le card devono essere DONE
listprojects                        mostra lista dei progetti di cui l'utente fa parte
showmembers [project name]          mostra membri di un progetto
addmember [project name] [username] aggiunge un nuovo membro al progetto
showcards [project name]            mostra le card relative ad un progetto
showcard [project name] [card name] mostra le informazioni relative ad una card
addcard [project name] [card name] [card description] aggiunge una nuova card al progetto
movecard [project name] [card name] [old state] [new state] cambia lo stato di una card
getcardhistory [project name] [card name] mostra la cronologia di una card
listusers                           mostra la lista di tutti gli utenti registrati
listonlineusers                     mostra tutti gli utenti attualmente online
readchat [project name]             mostra i messaggi della chat di un progetto
sendchatmsg [project name] "message" manda un messaggio nella chat in un progetto (il messaggio deve esser compreso tra ")
>
```

Comandi per esecuzione:

Javac -cp .:../lib/* *.java

Java -cp .:../lib/* ServerMain

Javac -cp .:../lib/* ClientMain

Tutto ciò va eseguito posizionandosi prima nella cartella src.