UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

Relazione progetto Worth

Reti di calcolatori

Prof.ssa Laura Ricci
Prof.ssa Federica Paganelli

Giulia De Paola
560622 - Corso A

Anno Accademico 2020/2021

Contents

Descrizione generale

WORTH è un sistema client/server per la gestione di progetti collaborativi basato sulla metodologia Kanban. Un progetto è caratterizzato da un nome univoco, una serie di card che rappresentano i compiti da svolgere per portarlo a termine e una lista di membri del progetto; questi ultimi rappresentano utenti registrati alla piattaforma, che hanno i permessi per operare sul progetto e utilizzarne i relativi servizi. Tramite interfaccia a riga di comando (CLI), l'utente interagisce con il client, il quale comunica con il server utilizzando una connessione TCP per tutti i comandi fatta eccezione per: *registration e listUsers, listOnlineUsers*, che avvengono rispettivamente mediante RMI e callback RMI. La chat, infine, è implementata tramite UDP multicast, ogni progetto infatti alla creazione avrà anche un ip univoco associato, per garantire che abbiano accesso alla chat solo gli effettivi membri del progetto.

Introduzione

Il progetto è stato sviluppato con il supporto dell'IDE *IntelliJ IDEA*, utilizzando il jdk 10.0.2. Si struttura in 3 package, quali:

- Client: contenente le classi
  - *ClientMain* dalla quale startare il client; è costituito da:
    - reply e sReply: variabili che conterranno la risposta del server;
    - username una stringa che rappresenta l'username dell'utente;
    - IPChat una ConcurrentHashMap che mantiene le associazioni tra gli ip multicast e i threads delle chat;
  - *NotifyClient:* è l'interfaccia per la callback, comprende i metodi *notifyRegistration*, notifica la registrazione di un utente agli utenti connessi; *notifyStateUpdate*, notifica il cambiamento di stato di un utente agli utenti connessi;
  - *NotifyClientImpl:* è la classe che implementa l'interfaccia *notifyClient* ed estende la classe RemoteObject. Oltre ad implementare i metodi dell'interfaccia *NotifyClient*, implementa i metodi *listUsers, listOnlineUsers* per restituire ai client che ne fanno richiesta le liste aggiornate rispettivamente degli utenti con il relativo stato, e degli untenti online;
  - *Chat:* estente la classe *Thread* si occupa della gestione effettiva della chat;
  - *Queue:* classe per la gestione della coda dei messaggi ricevuti nelle chat di progetto.
- Server che racchiude le seguenti classi:
  - *ServerMain* per l'implementazione del servizio RMI, la restore dei dati, e la gestione dei thread attraverso una fixed threadpool. In particolare la gestione delle richieste dei client è però a carico della classe CommandHandler;
  - *RegistrationImplementation* che implementa l'interfaccia RegistrationServer ed estende la classe RemoteServer la quale fornisce le funzioni necessarie per creare ed esportare oggetti remoti.
  - *CommandHandler* si occupa della gestione delle richieste del client attraverso l'utilizzo di due CopyOnWriteArrayList per la gestione di utenti e progetti, resi poi persistenti attraverso la scrittura su file;
  - *User* è la classe che rappresenta l'entità utente, caratterizzata dai campi:
    - *nickname:* una stringa che rappresenta l'username utente;

    - *password:* una stringa che insieme al nickname permette l'accesso alla piattaforma per gli utenti registrati;
    - *isOnline:* è una stringa che rappresenta lo stato di un utente che può essere online/offline. Questo campo è transient perchè non è necessario memorizzare lo stato utente nel file, pertanto non viene serializzato;
  - *UserDB* è la classe contenente i metodi per la gestione della persistenza dei dati utente. È costituita da un CopyOnWriteArrayList di User per effettuare operazioni sugli utenti. Contiene infatti i metodi per la lettura e scrittura da/su file attraverso la libreria gson.
  - *Project:* classe utilizzata per la rappresentazione dell'entità progetto. Un progetto è caratterizzato da:
    - *pName:* è la variabile che rappresenta il nome del progetto;
    - *projectMembers:* un CopyOnWriteArrayList di stringhe per la memorizzazione degli utenti;
    - *toDoLIst:* un CopyOnWriteArrayList di Card che rappresenta la lista delle card che si trovano nella fase todo;
    - *inProgressList:* un CopyOnWriteArrayList di Card che rappresenta la lista delle card che si trovano nella fase in progress;
    - *toBeRevisedList:* un CopyOnWriteArrayList di Card che rappresenta la lista delle card che si trovano nella fase to be revised;
    - *doneList:* un CopyOnWriteArrayList di Card che rappresenta la lista delle card che si trovano nella fase done;
  - *Card:* è la classe che rappresenta l'entità card; È caratterizzata dai seguenti attributi:
    - *cName:* la stringa che rappresenta il nome della card;
    - *cDescription:* la stringa che rappresenta la descrizione assiociata alla card;
    - *CardList:* un'enum costituita dai campi TODO, INPROGRESS,TOBEREVISED,DONE
    - *history:* un CopyOnWriteArrayList di CardList per tener traccia del flusso di lavoro della card;

- StoredData contiene:
  - *dataUsers.json* file json per la memorizzazione degli utenti e la sottodirectory;
  - *ProjectDir* costituita da:
    - todoList, inProgressList, toBeRevisedList, doneList, quattro file json per la persistenza delle liste. In particolare ogni lista, conterrà tutte la card nello stato rappresentato dalla lista, indipendentemente dal progetto di appartenenza;
    - *ip* file json contentente gli ip già utilizzati;
    - una directory per ogni progetto, caratterizzata dal nome del progetto. Inoltre ogni directory di progetto è caratterizzata da un file json per ogni card del progetto e un file json contentente la lista dei menmbri del progetto.

Dettagli implementativi

Il server è stato realizzato multithreaded, è infatti in grado di gestire richieste provenienti da più client contemporanemente. Per ottimizzare l'utilizzo della memoria e del processore e diminuire l'overhead di gestione dei thread, è stato implementato un FixedThreadpool, con 30 come soglia massima di thread creati. Questa scelta è stata effettuta per avere un giusto compromesso nei criteri di valutazione delle prestazioni del server. Lo scambio principale dei messaggi avviene sulla connessione TCP e la comunicazione tra client e server si basa sul paradigma richiesta-risposta. In paritcolare il client si connette alla porta 8000 sulla quale è in ascolto il server. L'accettazione della richiesta di connessione da parte di un client comporta la creazione, da parte del server, di un task di tipo *CommandHandler* la cui gestione è a carico del threadpool.

0.1 Vincoli

Per il corretto funzionamento del sistema sono presenti alcuni vincoli da rispettare, quali:

- prima di poter svolgere qualsiasi operazione è necessario che un utente abbia effettuato il login alla piattaforma, in caso contrario qualsiasi comando l'utente digiti diverso dalla register o login restituirà un messaggio di errore;
- il nome dell'utente, del progetto e della card non può contenere spazi quando si effettuano rispettivamente una register, createProject, addCard;
- per effettuare lo spostamento di una card sono rispettati i seguenti vincoli: $todolist \rightarrow progress$, $progress \rightarrow done \| revised$, $revised \rightarrow progress \| done$;
- utenti, progetti e card devono avere nome univoco;
- i messaggi della chat possono invece contenere spazi inoltre, per ogni read effettuata da un client, gli altri client diversi da questo leggeranno n-1 messaggi.

La funzione listProjects(), inoltre, per completezza stamperà la lista dei progetti di cui l'utente è membro nel formato: nomeProgetto@ipProgetto, quindi non solo il nome del progetto.

Server

1 ServerMain

All'avvio, il server ripristina lo stato del sistema, in particolare richiama le funzioni restoreUsers() e restoreProjects() che rispettivamente ripristinano gli utenti registrati e i progetti con le relative cards. La restoreUsers legge il file dataUsers.json contenuto nella cartella StoredData e aggiorna la struttura *userList*; la restoreProjects(), invece scorre i file contenuti nella cartella ProjectDir (contenuta nella cartella StoredData), e aggiorna la struttura *projects*. Infine il risultato delle operazioni di restore sarà stampato a schermo. Successivamente inizializza il servizio RMI che permette ai client di effettuare le registrazioni dei nuovi utenti e segnalare attraverso le callbacks i cambiamenti di stato degli utenti (che effettuano operazioni di login/logout/exit) dal sistema. Il server crea quindi l'istanza dell'oggetto remoto definito dalla classe *RegistrationImplementation*, effettua l'esportazione dell'oggetto stub di tipo *RegistrationServer* e crea il registry nel quale pubblica lo stub sulla porta 6500. Infine viene effettuata l'attivazione dei thread del servizio attraverso una fixedThreadpool di dimensione 30 thread. Viene creata la ServerSocket e si entra nel loop del server che attraverso un thread esegue il task centrale *CommandHandler* al quale passa il socket, la struttura contenente gli utenti, la struttura contenente i progetti e l'oggetto remoto.

2 CommandHandler

È la classe che si occupa della gestione effettiva delle richieste da parte dei client. Nel loop vengono creati il BufferedReader e il PrintWriter per la comunicazione da e verso i client e dopo aver controllato che il client non abbia inviato un messaggio vuoto o nullo si entra nello switch che per ogni comando controlla che l'utente sia loggato alla piattaforma e per ogni operazione verifica che l'utente abbia i permessi per effettuare l'operazione richiesta.

2.1 Protocollo Client-Server

Per alleggerire il traffico tra client e server è stato implementato un protocollo secondo il quale il server invia al client un codice che indica l'esito dell'operazione come descritto in seguito:

- 200: operazione andata a buon fine;
- 400: utente/progetto/card esiste già;
- 401: password errata;
- 402: l'utente non ha accesso al progetto sul quale vuole operare;
- 404: utente/card non trovato/a;
- 405: utente non loggato;
- 406: problemi con la moveCard, violati vincoli di spostamento card;
- 408: problemi con la moveCard, lista sorgente non contiene la card da spostare;
- 410, 411, 412: errore di sintassi nella digitazione del comando;
- 413: utente già membro del progetto;
- 414: errore con la cancelProject, quando si prova a cancellare un progetto anche se non tutte le sue cards sono nella doneList;
- 505: utente non registrato;
- 506, 507: errore nella scrittura del file json.

Tale protocollo è utilizzato per tutti i comandi eccetto:

- register che è implementato tramite RMI;
- showMembers che restituisce una stringa con i membri del progetto;
- listProjects che restituisce una stringa con i progetti associati a un utente;
- showCard che restituisce una stringa con le informazioni relative a una card;
- showCards che restituisce una stringa di tutte le cards;
- getCardHistory che restituisce una stringa con la history della card.

3 User

Nella classe user è stato sovrascritto il metodo equals in modo da confrontare gli utenti in base al nickname; è stato inoltre implementato il metodo *getIpFromPrj* che restituisce l'ip relativo a un progetto, null altrimenti.

4 UserDB

Implementa i metodi:

- readFile(), utilizzato per la deserializzazione degli utenti dal file *dataUsers.json*;
- readFromFile(), implementa le stesse funzionalità di readFile ma restituisce la struttura userData (metodo utilizzato dal server durante la restore);
- writeFile(CopyOnWriteArrayList <User> userList), utilizzato per la serializzazione, della struttura utenti passata come parametro, nel file *dataUsers.json*;

- getUsers(), restituisce un utente se contenuto nella struttura userData, null altrimeti;
- getUserProjectList(), prende la lista dei progetti di un utente li restituisce sotto forma di stringa;
- getPrjMembers(), metodo che restituisce la stringa contenente i membri di un progetto
- isMember(String projName, User usr), metodo che controlla la presenza di un progetto nella lista progetti dell'utente;
- foundUser(User usr), metodo che cerca un utente della struttura userData;
- deleteProject(User u, String prjName), implementa la rimozione di un progetto dalla lista dei progetti di un utente aggiornando il file *dataUsers.json* attraverso una scrittura;
- addMemberL(String projName, User usrDaAggiungere, User usrCheAggiunge), si occupa dell'aggiunta di un utente alla lista di membri del progetto, e successivamente aggiorna anche il file degli utenti inserendo il progetto, attraverso la funzione addProject;
- addProject(User u, String projectName), metodo per aggiungere un progetto a un utente, dopo aver verificato che l'utente sia presente nella struttura, viene aggiunto il progetto e aggiornato il file *dataUsers.json* attraverso una scrittura su file;

5 Project

Implementa i metodi:

- getIp() per recuperare l'ip associato a un progetto;
- getProjectName() restituisce il nome di un progetto;
- getAllCards() restituisce la stringa con tutte le card;
- getProjectMembers() restituisce la lista dei membri di un progetto;
- AddMember(String user) aggiunge un membro a un progetto, in particolare alla struttura projectMembers;
- readAllList() deserializza le liste todoList.json, inProgressList.json, toBeRevisedList.json, doneList.json
- writeAllLists(String path) serializza le liste di cards todoList.json, inProgressList.json, toBeRevisedList.json, doneList.json
- readProjectMembers(String path) deserializza il file projectMembers.json
- readToDoL() si occupa della deserializzazione del file toDoList.json
- writeProjectMembers(String path) serializza la struttura projectMembers e restituisce l'esito dell'operazione;
- foundProjCards(String prjName) utilizzato per il controllo dell'esistenza di una card in un progetto, restituisce l'array di card del progetto;
- writeCardFile(String prjName, String cardName, Card card) metodo utilizzato alla creazione di una card, si occupa di renderla persistente creado il file relativo ad essa;
- writeToDoL() si occupa della serializzazione della lista toDo nel file toDoList.json;
- writeDoneL() si occupa della serializzazione della lista done nel file doneList.json;

- addCard(Card c) metodo per aggiungere una card a un progetto;
- moveCard(String cardName, String srcList, String destList) metodo per lo spostamento di una card, trova la lista sorgente e destinataria della card, verifica che non siano null e che siano diverse; controlla i vincoli di spostamento e se sono rispettati effettua l'operazione. Se la lista sorgente non contiene la card che si vuole spostare restituisce 0; se i vincoli non sono rispettatiti restituisce -6; se le list sono null restituisce -1, 7 se l'operazione va a buon fine;
- existenceCardexistenceCard(String cardName) controlla l'esistenza di una card
- getCardInfo(String cardName) restituisce tutte le informazioni relative a una specifica card;
- getCard(String cardName) ottiene e restituisce il riferimento alla card passata se esiste, null altrimenti;
- deletePrjDir(File dir) metodo per la cancellazione della directory del progetto e del suo contenuto.
- void compareIP(), boolean storeIP() utilizzati rispettivamente per la deserializzazione e serializzazione degli indirizzi ip da e nel file *ip.json*

Il costruttore *Project(String user, String prjName)* è utilizzato alla creazione di un progetto per la generazione di un ip associato a ques'ultimo. In particolare creo un ip multicast a partire dall'indirizzo "239.255.x.y" dove x e y sono due numeri casuali compresi fra 0 e 255. Il riuso è garantito dal fatto che gli ip utilizzati vengano salvati in un file *ip.json* nella cartella ProjectDir. Una volta generato l'ip infatti verrà confrontato con quelli già utilizzati, attraverso il metodo *boolean compareIP(String ip)* e se dovesse essere uguale viene scartato, (se ne genera un'altro), altrimenti può essere utilizzato, quindi sarà assegnato al progetto appena creato. Alla cancellazione di un progetto verrà quindi rimosso il suo ip dal file, in modo tale che se dovesse essere generato identico in una prossima createProject questo potrà essere riutilizzato.

6 Card

Oltre a set e get degli attributi implementa i metodi:

- set_History(String listaDest) che aggiunge un nuovo stato alla history in base alla lista di destinazione;
- getLastElement(String cardName), l'ultimo elemento della history list rappresenta la lista corrente in cui si trova la card;

È stato inoltre sovrascritto il metodo equals(Object o) per poter confrontare gli oggetti di tipo cards confrontando il loro nome.

Client

1 ClientMain

All'avvio, (si presuppone che il server sia attivo) il client inizializza le proprie strutture dati; viene recuperata la reference all'oggetto remoto chiamato *serverStub*; viene stabilita una connessione TCP con il server, utilizzata per tutte le richieste. Oltre ai metodi richiesti da specifica sarà sempre possibile utilizzare il comando exit per terminare l'esecuzione del client. In caso di un errore di sintassi nella digitazione verrà stampato un messaggio di errore e invitato l'utente a inserire nuovamente il comando dopo aver utilizzato l'help per consultarne la sintassi. Alcuni controlli sulla sintassi sono effettuati lato client, prima di inviare i comandi al server, per alleggerire il carico del traffico. Tutti i comandi faranno richieste al server eccetto *listUsers, listOnlineUsers* che utilizzano l'oggetto remoto *callbackObj* di tipo *NotifyClientImpl* per tenere aggiornata la struttura locale degli utenti registrati e online. Infine continete un metodo per l'interpretazione dei messaggi del server e un metodo statico *readMSG(String ip)* per la lettura dei messaggi da una chat.

2 Chat

Gli ip multicast del progetto sono comunicati tramite il servizio RMI. In particolare a seguito di una createProject o una addMember verrà richiamato tramite l'oggetto callbackObj il metodo remoto notifyaddToProject(String ip) che si occupa della creazione di un thread per la gestione della chat del progetto identificato dall'ip passato. La classe Chat si occupa dell'effettiva gestione della chat. Crea ed invia il datagramPacket da inoltrare ad un gruppo multicast, si occupa della ricezione dei datagramPacket su un gruppo di multicast popolando la coda dei messaggi, contiene inoltre il codice che i vari thread eseguono per stare in ascolto nella chat di un progetto. La start di una Chat esegue la join al gruppo multicast e finchè non viene interrotto resta in attesa di ricevere messaggi sulla multicastSocket, se li riceve popola la coda messaggi della chat progetto. È stato inoltre implementata una classe Queue per la gestione delle operazioni sulla coda di messaggi della chat. Si occupa della lettura dei messaggi contenuti nella coda e della scrittura di un messaggio in chat. Rispettivamente il metodo readClear() si occupa della lettura e svuota la coda dopo la prelevazione dei messaggi; il metodo put(String msg) aggiunge il nuovo messaggio in coda.

RMI service

Il servizio RMI viene utilizzato per la registrazione degli utenti alla piattaforma e per la registrazione degli utenti al servizio di notifica attraverso il quale il server avvisa gli altri client della registrazione di un nuovo utente o del cambiamento di stato (online/offline) di un utente. All'avvio del server avviene l'esportazione dello stub, la creazione del registry e la registrazione dello stub nel registro come 'ServerRMI'. Il client sarà quindi in grado di ottenere una reference allo stub ed invocare i metodi contenuti nell'interfaccia *RegistrationServer*, implementati nella classe *RegistrationImplementation*, ovvero:

- register(String username, String password) meodo utilizzato dal client per la registrazione di un nuovo utente al sistema;
- registerForCallback(NotifyClient client, String user) permette la registrazione di un nuovo utente al servizio di notifica, uno dei parametri passati alla funzione è infatti la reference allo stub del client esportato. Ogni utente dopo essersi loggato effettuerà una registrazione al servizio di callback tramite questo metodo;
- unregisterForCallback(String user) permette la deregistrazione del client dal servizio di notifica tramite callback. Utilizzato a seguito del logout dell'utente.

Inoltre implementa i metodi:

- notifyRegistratio(String username) che permette di segnalare ai client di aggiornare la lista degli utenti registrati a seguito di una nuova registrazione, chiamando il metodo notifyRegistration sull'oggetto definito dalla classe del client NotifyClient;
- logIn(String username)/logOut(String username) che analogamente segnalano ai client di aggiornare la propria struttura locale a seguito rispettivamente di una login e una logout da parte di un utente richiamando il metodo notifyUserStateUpdate che a sua volta chiamerà il metodo notifyUserStateUpdate sull'oggetto definito dalla classe del client NotifyClient;

Lato client invece è stata implementata la classe NotifyClientImpl che estende RemoteObj implementa l'interafaccia NotifyClient. Questa classe implementa i metodi utili per le callback tenendo aggiornate le strutture dei singoli utenti. In particolare:

- notifyRegistration(String username, String state, ConcurrentHashMap l) a seguito di una nuova registrazione, aggiorno la struttura locale del client;

- notifyUserStateUpdate(String username, String state, ConcurrentHashMap list) notifico il cambio di stato($online \rightarrow offline$ / $offline \rightarrow online$) di un utente;
- void notifyaddToProject(String ip) crea il thread per la gestione della chat del progetto il cui ip è passato come parametro;
- notifyCancelProject(String ip) interrompe il thread che gestisce la chat del progetto passato come parametro;
- listUsers(), listOnlineUsers() sono i metodi che tengno aggiornate le strutture dei client rispettivamente per gli utenti registrati alla piattaforma e gli utenti online al momento della richiesta.

Strutture dati e concorrenza

Per gestire al meglio la concorrenza le principali strutture dati utilizzate, successivamente descritte, sono thread-safe, mentre sono state rese **synchronized** le funzioni per la lettura/scrittura da/su file:

- **CopyOnWriteArrayList** questa scelta è stata fatta in quanto la scrittura di tale struttura provoca la creazione di una copia dell'intero array sottostante. L'array originale viene mantenuto, in modo che il lettore possa leggerlo in modalità sicura, mentre la copia dell'array viene modificata. Terminata la modifica un'operazione atomica scambia il vecchio array con quello nuovo affinchè le lettture successive abbiano accesso alle nuove informazioni.
- **ConcurrentHashMap** scelta perchè supporta la concorrenza completa dei recuperi e un'elevata concorrenza per gli update. Le operazioni di recupero (incluso get) generalmente non si bloccano, quindi possono sovrapporsi alle operazioni di aggiornamento (incluso put e remove).

Compilazione

Per la serializzazione e deserializzazione dei file JSON è stata utilizzata la libreria esterna di Google **Gson** (https://github.com/google/gson), occorre quindi scaricare la versione 2.8.6. Tuttavia il file .jar necessario per la compilazione è già presente dell'archivio del progetto. E' consigliato compilare ed eseguire il progetto in un IDE come ad esempio Intellij. Se si utilizza Intellij dopo aver estratto la cartella dall'archivio 560622_DePaola_Giulia occorre importare la libreria esterna sull'IDE ed eseguire prima la run della classe ServerMain e successivamente la run di uno o più ClientMain. In alternativa i comadi per la compilazione sono:

```
javac -cp ./gson-2.8.6.jar -d bin/ src/Server/*.java src/Client/*.java
```

per l'esecuzione del server:

```
java -cp bin Server.ServerMain
```

per l'esecuzione del client:

```
java -cp bin Client.ClientMain
```