Università di Pisa

Relazione per il progetto di reti WORTH

Samuele Iacoponi - 531852 - corso B

A.A 2020/2021

Indice

Capitolo 1

Introduzione

Relazione per descrivere il progetto di reti

1.1 Descrizione del problema

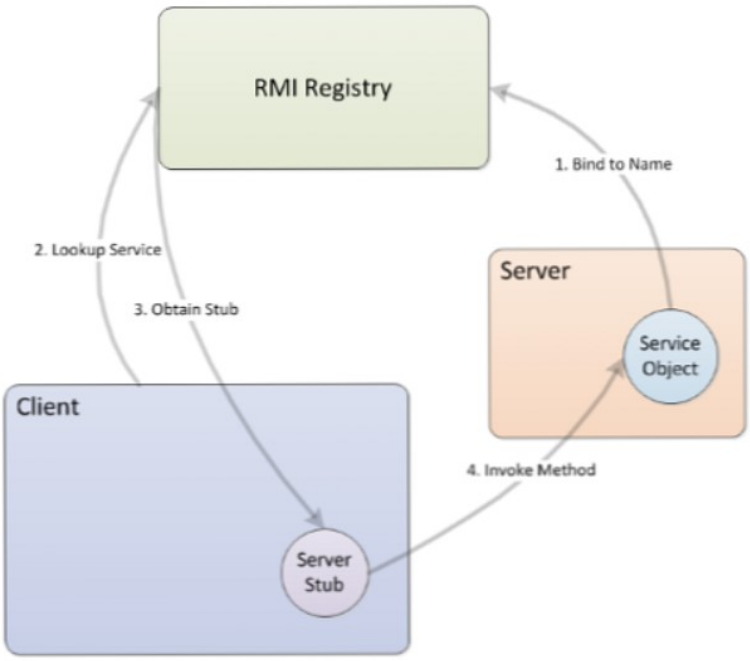
Il progetto consiste nell'implementazione di un programma che sfrutta la metodologia kanban. Tramite il metodo Kanban si rovescia il punto di osservazione e si concepisce quello produttivo come un processo che va da valle a monte in cui si svolgono le attività necessarie solo nel **momento in cui ce n'è effettivamente bisogno.** [2]

1.2 Scelte progettuali

Il server ho deciso di renderlo multithread invece di utilizzare il multiplexing dei canali con NIO. Ho scelto anche di utilizzare una **Command Line Interface (CLI)** al posto di una **GUI** usando uno switch all'interno del server e del client dove gestisco tutti i possibili comandi che il progetto comprende.
Dato che il server deve essere **persistente** sui progetti e sugli utenti, per garantire questa proprietà ho salvato le strutture all'interno di vari file **.json** che il server andrà a creare nella prima esecuzione oppure andrà a recuperare e leggere ogni volta che verrà riavviato.

1.2.1 RMI e Callback

Per registrare un nuovo utente uso il metodo **RMI** (Remote Method Invocation) cioè un insieme di meccanismi per permettere ad un'applicazione java in esecuzione su una macchina locale di invocare i metodi di un oggetto situati su un'altra macchina remota.
Per implementare questo sistema utilizzo un'interfaccia che estende *Remote* chiamata ***RMIregisterInterface***.
L'invocazione di questi metodi è possibile grazie alla creazione di un **registro**, cioè un servizio di *naming* che associa nomi di server alle loro referenze, il registro deve essere facilmente localizzabile. Il client recupera il riferimento tramite *lookup*. I metodi disponibili sono dichiarati nell'interfaccia.



In caso di cambiamento di stato il server notifica gli altri utenti presenti grazie alla ***callback***, un sistema di notifica asincrona. Questo è compreso nei metodi dell'oggetto remoto che il server fornisce.
Per ogni cambiamento di stato viene invocato il metodo **update** per andare ad effettuare le callback.
I cambiamenti di stato comprendono:

- cambio di stato da parte di un altro utente
- aggiunta ad un progetto
- cancellazione di un progetto

Strutture dati

Ho utilizzato diverse **strutture dati**, una per ogni *dato* da gestire tra cui:

- una per gli utenti dove vengono specificati username password e stato(online,offline)
- una per le cards usata per memorizzare il nome, la sua descrizione e una lista che contiene la *storia* dei suoi spostamenti
- una per i progetti dove vengono specificati il nome, le card che contiene, i membri e tutte le liste del percorso che una card può seguire

1.2.2 Consistenza

Per le operazioni di serializzazione/deserializzazione e per garantire la **consistenza**, cioè che i dati degli utenti e dei progetti siano mantenuti e quindi effettivamente utilizzabili, ho optato per salvarli in dei file .json, usando la libreria ***GSON*** per convertire oggetti java in json.

Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson is an open-source project Gson can work with arbitrary Java objects including pre existing objects that you do not have source code of.[1].
Ho creato un file per ogni **card** contente al suo interno la sua descrizione, che andrà ad essere aggiunto nella cartella del progetto.
Anche per gli indirizzi multicast sono andato a generare un file in modo da tenere in memoria l'ultimo indirizzo utilizzato andandolo ad aggiornare ad ogni creazione di un nuovo progetto.
All'avvio il server va a controllare se i file .json sono già presenti, nel caso in cui lo siano, va a leggerli e ad inserire i dati nelle relative hashmap, altrimenti andrà a crearli.

1.2.3 Gestione della concorrenza

Avendo creato il server in multithread ho implementato una classe *LoggedInHandler* che va ad implementare *Runnable*. A questa classe viene passata la lista degli utenti, la classe Server e le informazioni di connessione riferite al client. Successivamente, nel metodo run viene invocato il metodo start, contenente tutti i metodi che il client può digitare, **quindi per ogni client viene avviato un nuovo thread.**
L'avvio di un nuovo thread è affidato a un **threadpoolexecutor**, ho scelto di utilizzare **newCachedThreadpool** che va a creare nuovi thread in base alle necessità, riutilizza quelli che non sono in uso e cancella i thread dalla cache che non vengono utilizzati da un determinato tempo.

RMI e concorrenza

Le chiamate sull'oggetto remoto non vanno a bloccare l'oggetto e più client possono accedere ai metodi simultaneamente, questo fa su che RMI sia multithread ma non threadsafe, per questo motivo ho dovuto ragionare se sincronizzare o meno i metodi dell'oggetto remoto.
Sono andato a sincronizzare la registrazione e la cancellazione alla callback, ma non ho sincronizzato il metodo **register** in quanto l'aggiunta alla concurrent hashmap l'ho fatta con il metodo putIfAbsent, in quanto *threadsafe*

Concorrenza su strutture dati

La concorrenza sulle strutture dati ho deciso di gestirla attraverso **concurrent hashmap**, queste, essendo threadsafe, permettono di poter scrivere senza bloccare l'intera map consentendo, quindi, l'esecuzione di altre operazioni su di esse.
Usando metodi come **putIfAbsent** mi ha permesso di evitare di sincronizzare interi metodi o blocchi di codice.
Ne ho usate 2, una per gli utenti e una per i progetti. Per gestire la concorrenza sulle **cards** ho sincronizzato la struttura nei metodi all'interno della classe *Project* in modo che i thread in attesa possano andare comunque ad eseguire la parte di codice non sincronizzato.

Metodi lato server

Sul server sono andato ad utilizzare la keyword synchronized dove il metodo andava a modificare la struttura, questo per garantire che i client non andassero a modificare la stessa struttura dati nel solito momento.

1.3 Interfacce

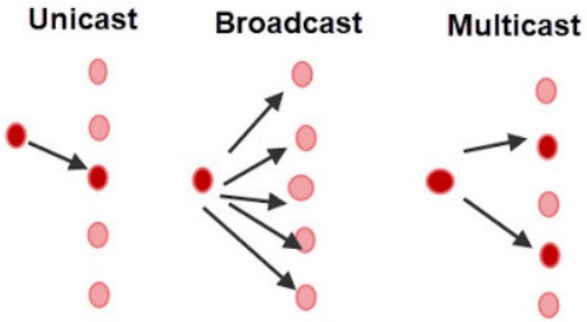
Per realizzare questo progetto sono andato ad utilizzare 3 interfacce:

- **NotifyInterface**
- **ServerInterface**
- **RMIRegisterInterface**

RMIRegisterInterface: questa va ad estendere **Remote** in quanto remote serve ad identificare quei metodi che possono essere invocati da una macchina remota, infatti, ogni oggetto remoto deve andare ad implementare questa interfaccia.
ServerInterface: con questa interfaccia dichiaro tutti i metodi che il server andrà ad implementare. viene implementata dalla classe *TCPServer*.
NotifyInterface: anche questa interfaccia estende **Remote** perchè dichiara i metodi invocati dal server per notificare un evento ad un client remoto. Questa interfaccia viene implementata dalla classe *ClientMain*

1.4 Generazione IP multicast

Per la gestione di invio/ricezione dei messaggi della **chat**, i client di uno stesso progetto si possono inviare i messaggi tra loro attraverso gruppi *multicast*, il cui indirizzo viene generato insieme alla creazione di un nuovo progetto. La generazione avviene attraverso il metodo *randomIP()* della classe *MulticastGen*, stando nel range di indirizzi $[224.0.0.0] - [239.255.255.255]$. Dopo la creazione del primo progetto a cui viene assegnato l'indirizzo 224.0.0.1 questo viene salvato nel file *MulticastIP.json*.
Alla creazione di un nuovo progetto, il server va ad aumentare di 1 l'indirizzo precedente salvato nel file, in modo che ogni progetto abbia un indirizzo multicast univoco.
Questo indirizzo poi deve essere noto a tutti i partecipanti del progetto.



Ogni volta che un utente viene aggiunto al progetto viene anche messo al corrente delle informazioni multicast attraverso **callback**.Quindi, i messaggi vengono inviati usando una connessione di tipo **UDP.**

1.5 Condizioni per cancellazione progetto

Un progetto può essere eliminato **solo** dopo che tutte le card sono state spostate nella liste *done*. Una volta che viene chiamata la *cancelproject* si controlla che la condizione sia verificata e successivamente il progetto viene rimosso dalla lista dei progetti, viene aggiornato il file .json e viene eliminata la directory con tutte le card al suo interno.
Viene fatto un ulteriore controllo, cioè che se la lista progetti è vuota dopo la cancellazione l'indirizzo IP viene resettato favorendone il riuso.

Capitolo 2

Descrizione programma

2.1 Descrizione classi principali

Le classi principali da cui avviare il programma sono

- **ServerMain:** dove vengono create le strutture dati, creato il registro per l'RMI e avviato la connessione TCP a cui vengono passati gli utenti e i progetti.
- **ClientMain:** questa classe consiste nel stabilire la connesione TCP con il server ed entrare in un loop infinito nel quale inviare comandi fino a quando non viene digitato il logout.
- **TCPServer:** Appena avviata la classe ServerMain viene richiamata anche questa classe che si occupa di aprire il socket per la connessione TCP e di creare oppure recuperare informazioni dai file .json.
  entra in un loop infinito dove attende connessioni dai client, successivamente costruisce le informazioni di connessione riferite al client per poi avviare il thread
- **LoggedInHandler:** Tramite le informazioni ottenute dalla classe ***TCPServer*** viene avviato il comando execute del **Threadpoolexecutor**, avviando il thread e, tramite il metodo **start()** gestire effettivamente i comandi inviati dal client fino a quando non viene effettuato il logout.

2.1.1 Altre classi

Le principale altre classi che ho utilizzato sono:

- **User:** Descrive la struttura dati per gli utenti
- **Project:** Descrive la struttura dati per i progetti
- **SignedUpUsers:** Memorizza gli utenti nel file **Users.json**.
- **SignedUpProjects:** Memorizza i progetti nel file **Projects.json**.
- **Card:** descrive la struttura dati per le card contenente le informazioni relative.
- **Multicastgen:** è la classe per generare un indirizzo multicast da associare al progetto per poi utilizzare la chat.

- **Login:** Ha il compito di salvare il risultato del login, aggiunge alla lista degli utenti il nuovo stato di online (nel caso l'operazione vada a buon fine) e aggiunge alla lista del multicast del progetto la segnalazione che lui è membro.
- **ToClient**$< T >$**:** Questa classe l'ho resa generica in quanto l'ho usata per rispondere con diversi tipi di oggetto al client.
- **UserAndStatus:** Usata per restituire la coppia username-status.
- **InfoCallback:** Classe usata per salvare le informazioni che verranno utilizzate per effettuare le callback.
- **ClientInfo:** Memorizza le info sulla connessione del client.
- **infoMultiCastConnection:** Memorizza le info sulla connessione multicast per i progetti

2.2 Modalità di esecuzione e funzionamento del programma

Per avviare il programma le 2 classi principali da compilare sono **ServerMain** e **ClientMain**.
Non sono necessari argomenti da passare al programma. Una volta avviato, il server, va a creare l'oggetto remoto e inizializzare i file degli utenti e dei progetti, dopo va a creare il registro e il riferimento ad esso al quale il client dovrà accedere tramite *lookup*. A questo punto il server avvia il metodo *TCPStart()* dove entra in un ciclo infinito in attesa di nuove richieste di connessione dei client, ottiene le informazioni e avvia un nuovo thread grazie al **threadpool**.
Il nuovo thread rimane in attesa di comandi fino a quando non viene effettuato il logout oppure viene lanciata un'eccezione.

Capitolo 3

Istruzioni sul programma

Per compilare correttamente il progetto ed eseguirlo è necessario scaricare una libreria esterna, ***GSON*** dal sito
https://search.maven.org/artifact/com.google.code.gson/gson/2.8.6/jar.
Una volta avviato il server e il client è possibile iniziare ad utilizzare il programma.
Tutti i comandi devono essere lowercase in quanto il programma è case-sensitive.
Come prima operazione è necessario registrare un nuovo utente questo è possibile attraverso il comando

register username password

dopo sarà necessario effettuare l'operazione di login digitando

login username password

Dopo sarà possibile cominciare a creare progetti ed usufruire degli altri comandi.
Alla fine della sessione è possibile uscire effettuando il logout con la sintassi

logout username

Gli altri comandi disponibili:

- ***listusers:*** per recuperare la lista degli utenti registrati con il loro stato
- ***listonlineusers:*** per recuperare la lista degli utenti registrati che sono online
- ***listprojects:*** per recuperare la lista dei progetti di cui l'utente loggato fa parte
- ***createproject projectName:*** questa comando crea un nuovo progetto avente come membro l'utente che ha richiesto la creazione
- ***addmember projectName Nickutente:*** aggiunge al progetto projectName l'utente Nickutente
- ***showmembers projectName:*** Recupera la lista dei membri del progetto projectName
- ***showcards projectName:*** recupera la lista delle cards associate al progetto projectname

- ***addcard projectName cardName description:*** aggiunge al progetto projectName una nuova card di nome CardName con una breve descrizione **da inserire senza spazi**
- ***movecard projectName cardName beginList endList:*** muove la card cardName nel progetto projectName dalla lista beginList alla lista endList, se soddisfa i vincoli di spostamento elencati qui sotto.

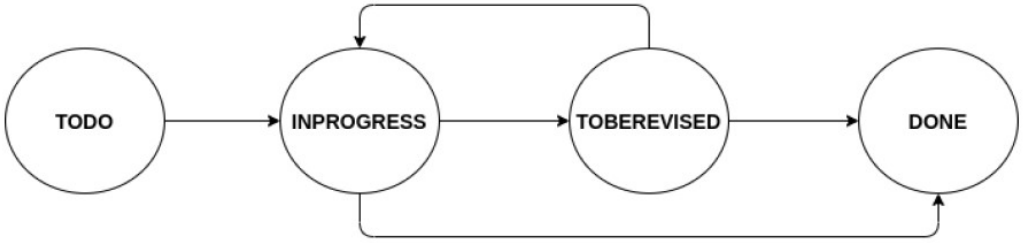


Figura 3.1: Vincoli spostamento cards

- ***getcardhistory projectName cardName:*** recupera la lista degli spostamenti della card cardName all'interno del progetto projectName
- ***readchat projectName:*** recupera i messaggi della chat riferiti al progetto projectName se disponibili
- ***sendchatmsg projectName:*** una volta lanciato il commando si potrà digitare il messaggio da inviare
- ***cancelproject projectName:*** se le card sono tutte finite sarà possibile eliminare il progetto

Bibliografia

[1] *Gson*. URL: https://sites.google.com/site/gson/gson-user-guide#TOC-Overview.

[2] *Metodo Kanban*. URL: https://www.makeitlean.it/blog/il-sistema-kanban-un-esempio.