Relazione progetto Laboratorio di Reti: WORTH

Marco Colavita

Gennaio 2020

Contents

Chapter 1

Introduzione

Scopo di questa relazione è quello di presentare il progetto per l'esame di Laboratorio di Reti: WORTH acronimo di WORkTogetHer. Il lavoro si focalizza sull'organizzazione e la gestione di progetti in modo collaborativo. Questi ultimi possono essere in generale qualsiasi attività che possa essere organizzata in una serie di compiti che sono svolti da membri di un gruppo. Il progetto consiste nell'implementazione di WORkTogetHer: uno strumento per la gestione di progetti collaborativi che si ispira ad alcuni principi della metodologia Kanban.
In WORTH, un progetto, viene identificato da un nome che è univoco ed è costituito da un insieme di *card*, che rappresentano i compiti da svolgere per portare a termine il progetto. Ogni progetto possiede una lista di membri partecipanti, ovvero degli utenti che possiedono i diritti per modificare/creare delle card ed accedere ai servizi associati al progetto, quali ad esempio la chat.
L'impostazione del progetto segue il paradigma di comunicazione di tipo *client/server* e prevede allo scopo due classi, $ClientMain$ e $ServerMain$ che rappresentanto rispettivamente il lato client ed il lato server di WORTH.

Chapter 2

Scelte progettuali

2.1 Strutturazione del codice

Per la progettazione, sono state realizzate diverse classi, cercando così di rendere indipendente ogni modulo, seguendo quanto più possibile il principio di SoC, ovvero di *Separation of Concerns*. A tal proposito è stata presa la decisione di tenere separati tre dei moduli fondamentali per la realizzazione di WORTH, ovvero:

- Modulo Utente;
- Modulo Progetto;
- Modulo Carta;

2.2 Classi

2.2.1 Utente

Tra i tre moduli principali sopracitati, la classe *Utente* è la classe che rappresenta l'unità più semplice modificabile. Serve per rappresentare e gestire un generico utente iscritto al servizio WORTH.
Tale modulo contiene le informazioni relative all'utente WORkTogetHer. Ogni utente in WORTH sarà identificato da:

- un *nome*, che sarà univoco all'interno del sistema;
- una *password*, nota solo al server WORTH per l'identificazione;
- uno *status*;

L'attributo *status* della classe utente serve per indicare lo stato dell'utente registrato al servizio, ovvero *online* ed $offline$. L'utente appena registrato avrà il proprio status ad offline, una volta effettuato il login il suo status verrà cambiato in online; viceversa, quando effettuerà il logout lo stato cambierà da online ad offline.

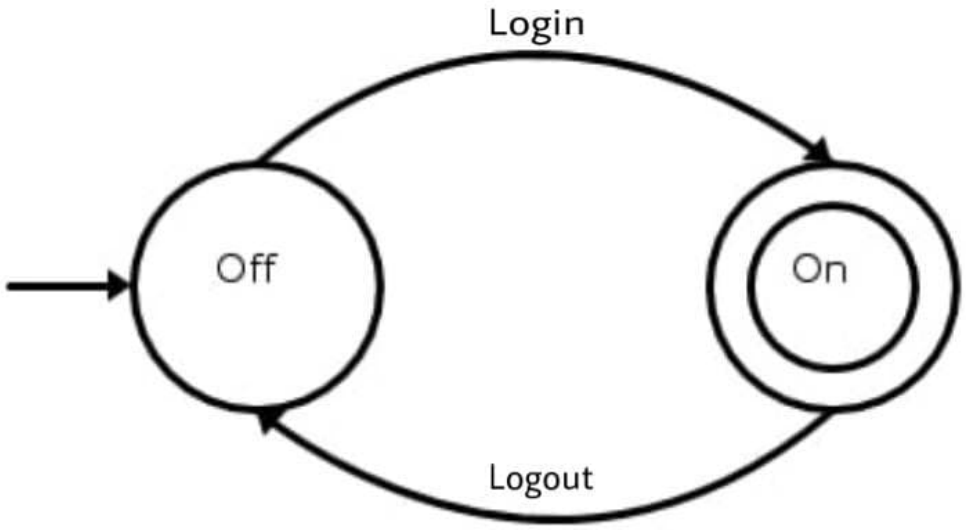

Figure 2.1: Stato dell'utente

Le informazioni rigurardanti la registrazione, il *login* e *logout* del generico utente verranno fornite, in seguito, all'interno della classe $ClientMain$.

2.2.2 Progetto

La classe progetto è la classe utilizzata per descrivere un progetto all'interno di WORTH.
Per la creazione di un nuovo progetto nel servizio, è fondamentale conoscere:

- il *nome* del progetto, che dovrà essere univoco
- il $nome - utente$ dell'utente che crea il progetto
- $indirizzo - multicast$ e *porta* che saranno necessari al ClientMain e ServerMain per connettersi al nuovo progetto e scambiare messaggi tra membri del progetto.

Ogni progetto conterrà delle card, ovvero dei task che dovranno essere portati a termine, ed in più una lista di membri del progetto che possono creare, e di conseguenza aggiungere al progetto, delle card che possono essere modificate. Il progetto alla creazione, ha associate quattro liste che definiscono il flusso di lavoro come passaggio delle card da una lista alla successiva: TODO, INPROGRESS, TOBEREVISED, DONE. Qualsiasi membro del progetto può spostare la card da un lista all'altra rispettando dei determinati vincoli che verranno descritti nel *paragrafo* 2.2.3. Tale spostamento è gestito utilizzando il metodo *moveCard*.
Ogni progetto, naturlamente, può essere cancellato e la sua cancellazione può avvenire con successo se e solo se tutte la cards che possiede si trovano nello stato DONE.

2.2.3 Card

Tale classe ha il compito di definire i task di un generico progetto che dovranno essere portati a termine. Ogni card sarà unica all'interno del progetto, verrà creata associandole un nome univoco ed una sua descrizione. Oltre a questi due attributi, la carta conterrà informazioni riguardanti il suo status (inizialmente settato a "TODO"), ovvero in che lista si trova, ed un proprio ciclo di vita, ovvero una storico contenente tutte le liste che la carta ha attraversato.

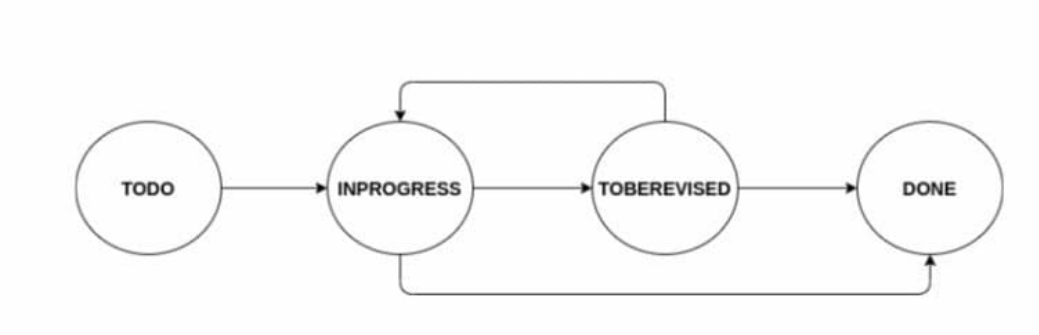

Figure 2.2: Ciclo di vita di una Card

La figura 2.2 è un automa a stati finiti che serve per specificare i vincoli che ci sono per lo spostamento di un task da una lista alla successiva. Come mostrato in figura, si può vedere che una carta dalla lista TODO può spostarsi solo verso la lista INPROGRESS, da quest'ultima può passare alla lista DONE che ne denota il completamento, oppure finire nella lista TOBEREVISED. Se la carta si trova nella lista TOBEREVISED potrà tornare

indietro verso la lista INPROGRESS oppure terminare e passare alla lista DONE.
Tutti i vincoli sullo spostamento sono garantiti dal metodo *moveCard* della classe *Progetto*.

2.2.4 ClientMain

Seguendo appunto un paradigma di comunicazione di tipo client/server è stato opportuno definire una classe per il client ed una per il server.
La classe $ClientMain$ implementa i metodi definiti all'interno dell'intrefaccia $RMIEventIntreface$, metodi che permettono al client di essere aggiornato sulle azioni di registrazione, login e logout, e callbacks di aggiornamento di informazioni delle connessioni Multicasting utilizzando il meccanismo RMI. Le callbacks che vengono ricevute dal client sono callbacks che servono o per andare ad aggiornare la lista degli utenti registrati al servizio o per aggiornare le informazioni riferite alle connessioni Multicasting, informazioni relative ai progetti. Una volta creato il thread Client vengono definite:

- una $RMI - PORT$ relativa al meccanismo di RMI del client;
- una $TCP - PORT$ e un *indirizzoServer* con i quali è possibile instaurare una connessione con il server;
- una lista di *utenti* che sono iscritti al servizio WORTH;
- una lista *multicastSockets* contentente le informazioni multicast (NomeProgetto,indirizzoIp,porta,socket) relative ai progetti attivi su WORTH dove il client è membro.

Una volta ottenute queste informazioni, il client può richiedere l'apertura di una connessione verso il server. Accettata la connessione, il client può richiedere uno o più servizi offerti da WORTH.
I comandi sono i seguenti:

da 2021-01-27 16-22-49.png

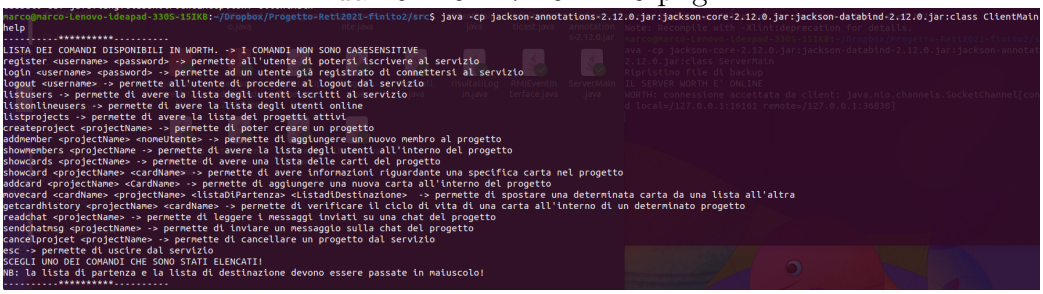

Figure 2.3: Lista dei comandi che il client può richiedere al server

Tutti i comandi verranno inviati tramite il protocollo di livello trasporto TCP al server il quale verifica la validità del comando e svolge le azioni necessarie al completamento del servizio richiesto. Gli unici comandi che il client non invierà al Server WORTH sono: *Register*, *listUsers* e *listOnlineUsers*. Gli ultimi due non saranno inviati poichè il client ha la possibilità di recuperare queste informazioni senza contattare il server.
Per quanto riguarda invece la registrazione, questa sarà affidata a RMI.
Il comando che il client dovrà necessariamente effettuare per primo è il comando *Register*. Una volta registrato, il client dovrà effettuare il login così potrà richiedere tutte le funzionalità che sono offerte dal Server WORTH.

2.2.5 ServerMain

La classe ServerMain serve per definire il server di WORTH.
Per realizzare la comunicazione con gli altri client è stato opportuno scegliere di utilizzare il multiplexing dei canali NIO mediante l'uso di un selettore, in modo da evitare l'overhead dovuti ai thread swithcing. Una volta avviato il server, viene creata una cartella *Backup* contenente due FILE di tipo JSON per memorizzare le informazioni su utenti iscritti al servizio e i progetti attivi. Il file *Utenti.json* conterrà la lista degli utenti che sono iscritti a WORTH, mentre *Progetti.json* la lista dei progetti attivi.
Il server a questo punto entra nella fase di multiplexing di richieste provenienti dai client. Di seguito è descritto, in maniera riepilogativa, come il server gestisce le richieste che accetta:

- Se la chiave fa riferimento ad un nuovo client che si connette per la prima volta, semplicemente si configura il socket relativo a questo client in modalità non bloccante (al fine di non sospendere l'attività del server durante le operazioni di I/O), viene costruita una HashMap che avrà come chiave il nuovo client e come valore un ArrayList che verrà uti-

lizzato per gestire gli esiti delle varie operazioni richieste ed infine si registra questo nuovo client in modalità OP_READ.

- Se la chiave è pronta per la lettura si effettua il parsing del comando ricevuto e si esegue l'operazione richiesta. In particolare, dopo che il server ha effettuato i controlli necessari per poter eseguire l'operazione il client viene registrato in modalità OP_WRITE e nell'ArrayList associato al client in fase di accept vengono inseriti gli esiti dell'operazione richiesta.

- Se la chiave è pronta per la scrittura allora vengono inviati gli esiti delle operazioni richieste. A questo punto il client viene registrato nuovamente in modalità OP_READ, in attesa di una nuova richiesta.

Allo starting, il server inizializzerà una lista per:

- Utenti;

- Progetti;

- Informazioni Multicast.

Come specificato nel paragrafo 2.2.2, ogni progetto ha bisogno di conoscere, oltre al nome, il suo indirizzo di Multicast e la porta. Quando un client richiederà la creazione di un nuovo progetto, sarà il server ad occuparsi di assegnare l'indirizzo Ip (utilizzando il metodo *generateIp*) e porta del nuovo progetto.
La lista di informazioni Multicast, invece, sarà passata al client non appena avrà effettuato il login, così da avere immediatamente le informazioni relative ai progetti ai quali è partecipe.

2.2.6 Classi Secondarie

Oltre alle classi sopra citate sono state utilizzate ulteriori classi al fine di supportare l'implementazione del proegetto. Queste classi sono:

- *statoUtente*: per associare un utente al proprio stato (ONLINE-OFFLINE);

- *risultatiLogin*: utilizzata durante la fase di login del client, al fine di consegnare le informazioni rigurdanti gli utenti di WORTH e dei progetti dove è membro;

- *Risultati*: per consegnare al client il risultato dell'operazione richiesta;

- *indirizziMulticast*: per associare le informazioni relative agli indirizzi Multicast (NomeProgetto,IndirizzoIp,Porta,Socket);
- $CallBackInfo$: per associare il nome dell'utente alla ClientInterface per la callback.

2.2.7 Strutture dati utilizzate

Per gesitre la concorrenza del client da parte del server, come specificato nel paragrafo 2.2.5, viene effettuato il multiplexing dei canali NIO mediante l'utilizzo di un selettore. Alla registrazione di un nuovo client viene utilizzata la struttura dati HashMap al fine di associare ad ogni client un ArrayList per la gestione degli esiti delle operazioni.
Oltre ad HashMap, le strutture dati che sono state utilizzate sono principalmente ArrayList, classe che implementa l'interfaccia $List < E >$.

2.3 Librerie esterne

Per la realizzazione del progetto WORTH è stato necessario installare delle librerie secondarie.
Per poter compilare il progetto, installare i file Jar:

- Jackson-core-2.12.0;
- Jackson-annotations-2.12.0;
- Jackson-databind-2.12.0

Chapter 3

Istruzioni per la compilazione

Il progetto è stato sviluppato su UBUNTU 20.04.1 utilizzando l'IDE IntelliJ. Le librerie esterne sono all'interno della directory contenente le classi del progetto. Eseguire da terminale i seguenti comandi (NB: prima di eseguire questi comandi spostarsi nelle cartella src contentente tutte le classi):

- Siccome tutte le classi si trovano nella stessa cartella, basterà digitare: **javac -d "class" -cp jackson-core-2.12.0.jar:jackson-databind-2.12.0.jar:jackson-annotations-2.12.0.jar *.java**
- A questo punto bisogna avviare il server WORTH: **java -cp jackson-core-2.12.0.jar:jackson-databind-2.12.0.jar:jackson-annotations-2.12.0.jar:class ServerMain**
- Infine eseguire il client: **java -cp jackson-core-2.12.0.jar:jackson-databind-2.12.0.jar:jackson-annotations-2.12.0.jar:class ClientMain**