WORTH

Antonio Marini

27 gennaio 2021

1 Architettura del progetto.

1.1 Architettura del server.

1.1.1 Inizializzazione e importazione dati.

Per prima cosa il server inizializza le sue strutture dati principali **UsersData** e **ProjectsData** che, rispettivamente conterranno informazioni e metodi utili per gli utenti e i progetti.

Nell'inizializzazione di UsersData viene chiamato un suo metodo interno per parsare, e quindi importare gli utenti salvati nel sistema.

Gli utenti sono contenuti nella directory *res/Users/* che contiene per ogni utente registrato a worth un file nominato *username.usr* che contiene un' istanza della classe **User** serializzata.

Anche i progetti, se esistono, vengono recuperati dalla cartella *res/Projects*, strutturata come da specifica, in modo da recuperare tutte le **Card** per ogni progetto. I dati relativi ai membri vengono ottenuti e ricostruiti a partire dai dati in **UsersData**. Gli indirizzi delle chat di progetto vengono ricreati da zero.

1.1.2 RMI: esportazione dell'oggetto remoto.

Viene creato un oggetto remoto **RegisterImpl** ed esportato nel registry sulla porta *9999*.

Servirà al client per invocare i metodi remoti **register** e **unregister** per registrare/deregistrare un utente, ma anche per i metodi per la registrazione alle callback **registerForCallback** e **unregisterForCallback**.

1.1.3 Attivazione dei thread del servizio.

Viene creato un **fixed** threadpool di dimensione 30 thread.

Si entra nel loop del server: viene creato thread che fa partire il task centrale **Service**, condividendo le risorse **UsersData** e **ProjectsData**.

1.1.4 Sincronizzazione e concorrenza.

I thread dei task **Service** che gestiscono i client condividono, le strutture dati relative agli utenti **UsersData** e ai progetti, **ProjectsData**.

I metodi che modificano il loro stato contengono blocchi synchronized, quindi due o più client non potranno mai modificare lo stato di un utente o di un progetto contemporanemante. I blocchi synchronized sono abbastanza brevi, e non

mantengono quindi la struttura bloccata per molto tempo.

Sebbene utilizzare meccanismi di lock a basso livello potrebbe migliorare le performance del programma, ho scelto di utilizzare meccanismi ad alto livello di lock implicite per maggiore leggibilità e manutenibilità del codice ma anche per le dimensioni ridotte del progetto.

1.1.5 Service.

Service è il servizio centrale del server worth; viene usato quando un client si connette alla connessione tcp creata dal server, inviando **username** e **password** e facendo iniziare la fase di login strutturata come segue:

1. **Check username e password**: Si manda indietro al client "yes", "no" oppure "duplicate"(già connesso) a seconda dell'esito. Se esito è positivo imposto lo stato dell'utente a online facendo anche una callback per avvisare tutti gli altri client connessi. Altrimenti la connessione termina (anche il thread di conseguenza).
2. **Invia indirizzi delle chat dei progetti.**
3. **Invia al client gli stati degli utenti**
4. **Registrazione del client per callback**

Terminata con successo la fase di login, inizia la fase dello scambio di comandi/dati tra client e server.

Il thread si blocca in attesa di leggere un comando con annessi argomenti inviati dal client.

Ricevuti il comando e gli argomenti viene chiamato **dispatchCommand()** che controlla la correttezza degli argomenti, esegue l'operazione relativa al comando ricevuto e restituisce al client un messaggio di esito dell'operazione.

Quando riceve comando di **logout**, il ciclo termina, l'utente torna offline, viene fatta la callback per notificare gli altri client connessi, e viene chiusa la connessione. Il thread termina.

1.2 Architettura del client.

Il client è stato implementato a riga di comando.

L'utente può digitare da stdin i comandi che verranno poi passati al server tramite la connessione tcp aperta nella fase di login. Tuttavia alcuni comandi che non hanno bisogno di essere mandati al server verranno direttamente eseguiti sul client. Per esempio la lettura della chat o l'invio di un messaggio.

1.2.1 RMI: advertisement stub client per ricevere callbacks.

Il client pubblica lo stub della sua interfaccia **NotifyInterface**, in modo che il server possa chiamare i suoi metodi per notificare tutti i client connessi quando un nuovo utente viene registrato oppure quando cambia stato da online a offline.

1.2.2 Register

Invoca il metodo remoto del server **RegisterImpl.register(username, password)**.

1.2.3 Login

Viene fatta una connect sulla porta in cui il server è in ascolto. Se il server invia esito positivo ("yes") il client riceve anche gli indirizzi delle chat dei progetti dell'utente con cui si è loggato, salvandoli nella struttura dati dedicata **ProjectsInfo**.

Vengono poi create le **ClientChat**, una per ogni progetto.

A questo punto il client potrà mandare tutti gli altri comandi eseguibili al server sulla connessione tcp.

1.3 Architettura della Chat.

1.3.1 Chat (server)

Il server si occupa, al momento della creazione o importazione di un progetto di generare un indirizzo multicast univoco per quella chat.

Il server non può leggere messaggi dalle chat ma può solo scrivere inviando messaggi di logging.

1.3.2 ClientChat

Un client che ha appena ricevuto gli indirizzi delle chat, (quando fa la login o quando crea un nuovo progetto) crea una **ClientChat** per ogni progetto.

1.3.3 ClientChatReader

Il costruttore di **ClientChat** crea anche un nuovo thread reader **ClientChatReader** che sta semplicemente in ascolto su una DatagramSocket per ricevere messaggi e metterli in una lista. Se una **ClientChat** viene chiusa viene fatto terminare anche il thread reader associato.

2 Compilazione ed Esecuzione

Compila i sorgenti del server e del client.

```
$ cd worth
$ javac -d bin/ src/server/*.java src/client/*.java
```

Esecuzione:

Esegui **prima** il server.

```
$ java -cp bin server.ServerMain
```

Esegui poi il/i client.

```
$ java -cp bin client.ClientMain
```

3 Utilizzo ed esempi di comandi.

Esegui il comando **help** all'interno del client per mostrare la lista di tutti i comandi disponibili con una piccola descrizione esplicativa del suo utilizzo. Di seguito viene presentata una completa spiegazione di tutti i comandi di worth attraverso un esempio.

3.1 Registrazione e Login

Registra un nuovo utente:
register *username password*

```
> register antonio 1234
```

Accedi a worth con utente registrato:
login *username password*

```
> login antonio 1234
< utente loggato con successo
```

Mostra stato degli altri utenti:
listUsers

```
(antonio)> listUsers
```

Mostra utenti online:
listOnlineUsers

```
(antonio)> listOnlineUsers
```

Quando hai finito esegui il logout:
logout

```
(antonio)> logout
```

3.2 Progetti e Cards

Vediamo un esempio di come utilizzare worth per creare un progetto *puliziecasa* condiviso con tutti i coinquilini (anche loro registrati su worth). Siano quindi daniele e gabriele due utenti già registrati su worth.

Creare un nuovo progetto:
createProjects *projectname*

```
(antonio)> createProject puliziecasa
```

Aggiungere nuovi membri a un progetto:
addMember *projectname username*

```
(antonio)> addMember puliziecasa gabriele
< gabriele e stato aggiunto al progetto puliziecasa
(antonio)> addMember puliziecasa daniele
<daniele e stato aggiunto al progetto puliziecasa
```

ATTENZIONE: un utente che viene aggiunto a un progetto deve prima fare il logout e riaccedere per poterlo visualizzare e ottenere i permessi.

Mostrare lista dei membri di un progetto:
showMembers *projectname*

```
(antonio)> showMembers puliziecasa
< antonio
  daniele
  gabriele
```

Chi fa parte del progetto ha i permessi per poterlo modificare.

Aggiungere una card:
addCard *projectname cardname (small description)*

```
(antonio)> addCard puliziecasa bagno Pulire bene il bagno!
< la card bagno e stata aggiunta al progetto puliziecasa
```

```
(daniele)> addCard puliziecasa cucina Pulire il piano cottura e
 il lavello.
< la card bagno e stata aggiunta al progetto puliziecasa
```

```
(gabriele)> addCard puliziecasa salone Lavare il pavimento.
< la card salone e stata aggiunta al progetto puliziecasa
```

Mostrare le cards:
showCards *projectname*

```
(antonio)> showCards puliziecasa
< bagno
cucina
salone
```

Mostrare info su una card:
showCard *projectname cardname*

```
(antonio)> showCard puliziecasa salone
< salone(todo) : #nome e stato
  Pulire il pavimento #descrizione
```

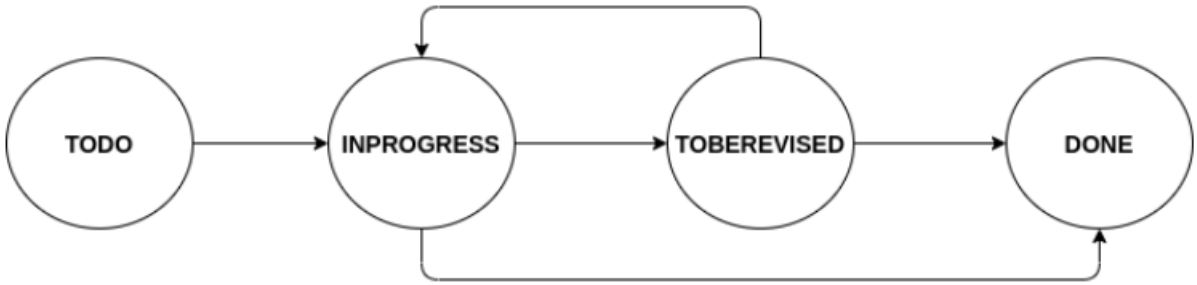
Inviare un messaggio sulla chat del progetto:
sendMessage *projectname message*

```
(antonio)> sendMessage puliziecasa io prendo la cucina allora
```

Leggere i nuovi messaggi della chat:
readChat *projectname*

```
(daniele)> readChat puliziecasa
antonio: io faccio la cucina allora
```

Una card può essere spostata seguendo il seguente schema:



I nomi delle liste sono:

- todoList
- inprogList
- reviseList
- doneList

Spostare una card tra le liste:
moveCard *projectname cardname sourceList destList*

```
(antonio)> moveCard puliziecasa cucina todoList inprogList #
todo -> inprogress
< la card cucina e stata spostata dalla todoList alla
inprogList

(antonio)> moveCard puliziecasa cucina inprogList doneList #
inprog -> done
< la card cucina e stata spostata dalla inprogList alla
doneList
```

Mostra storia di una card:
getCardHistory *projectname cardname*

```
(antonio)> getCardHistory puliziecasa cucina
< todo - inprog - done #stati in cui e stata la card
```

Quando tutte le card sono state messe nella lista done un membro qualsiasi del progetto può chiuderlo.

Chiudi un progetto:
removeProject *projectname*

```
(antonio)> removeProject puliziecasa
< progetto e stato eliminato correttamente
```