Progetto WORTH

progetto reti di calcolatori laboratorio 2020/2021

Fabio Bocci 580100

INDICE

1 Descrizione del progetto

1.1 Richieste del cliente

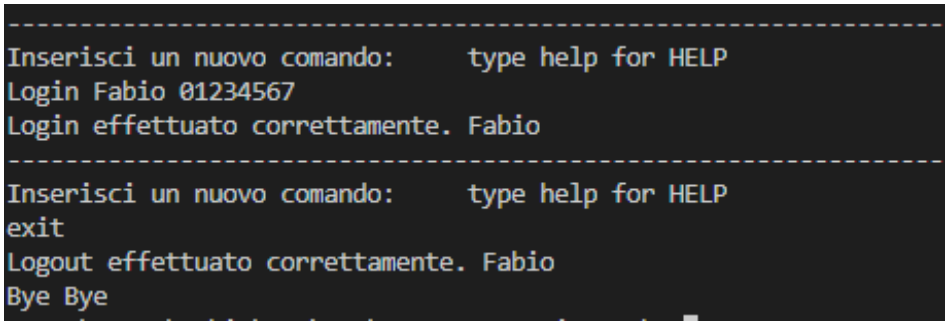
Il sistema WORTH è un'applicazione Client-Server based. Dove il server mette a disposizione comandi per la gestione di progetti a tutti i client connessi, che abbiano superato la fase di riconoscimento.

Ogni progetto è identificato da un nome univoco. Un progetto è costituito da un insieme di card, ovvero i task da svolgere per portar a termine il progetto, suddivise in quattro insiemi più piccoli a seconda del loro attuale stato (TODO - INPROGRESS – TOBEREVISED – DONE). Ad ogni progetto è associata una chat dove gli utenti online partecipanti possano usare per comunicare e leggere messaggi di sistema: quando una card cambia stato il server scriverà un messaggio di sistema per avvisare tutti i client l'avvenuto cambiamento.

Ogni card è composta da un nome unico all'interno del progetto, una descrizione testuale e uno stato, che serve per rappresentare l'attuale configurazione della card.

1.2 Introduzione all'implementazione

Il progetto è implementato lato client attraverso un'interfaccia testuale.



Per poter eseguire correttamente i comandi è stata creata una funzione help che trascrive a schermo tutti i comandi effettuabili dal client

Ogni comando non è case sensitive, ovvero non fa distinzioni fra maiuscole e minuscole, ad eccezione dei campi come: nome_utente, Password, NomeProgetto, NomeCard.

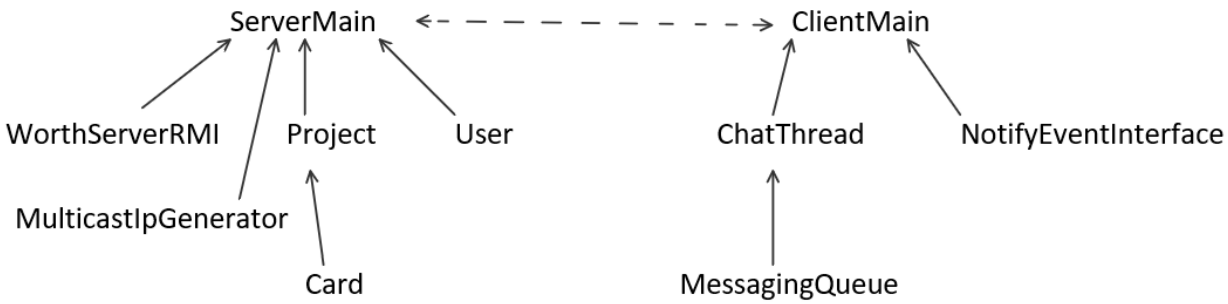
Se non si possiede un account, nome_utente e password propri, è possibile crearne uno utilizzando la funzione register (NB: la password è richiesta di essere di almeno sette caratteri lunga per motivi di sicurezza).

Sul client vengono effettuati anche controlli di sicurezza, ovvero se l'utente loggato può accedere al progetto richiesto, o se l'utente è loggato correttamente.

2 Implementazione

2.1 Classi create

L'implementazione del sistema WORTH è stata effettuata partendo dalla formazione di una gerarchia delle classi.



Quindi partendo dalle classi di più basso livello lato server:

Classe Card

La classe Card è la diretta implementazione delle card all'interno dall'applicazione. Contiene:

- **Nome** -> una String che funzione come ID all'interno del progetto;
- **Stato** -> implementato attraverso Enum di CardStatus per semplificare la lettura;
- **Desc** -> una String contenente la descrizione.

Per implementare le funzionalità del comando GetCardHistory ho dovuto creare:

- **CardHistory** -> una lista contenente tutta la storia degli stati della card dal momento della sua creazione, in poi.

Ho implementato tutta la classe basandomi sulle specifiche delle classi serializzabili in Json, in questo modo il server ha la possibilità di salvare tutti i dati in file *.json.

Classe Project

La classe Project è la diretta implementazione dei progetti all'interno dell'applicazione. Contiene:

- **Nome** -> una String che funziona come ID all'interno del server;

- **Cards** -> una lista contenente tutte le Card del progetto;
- **Cards_TODO** -> una lista di supporto per salvare i nomi delle card in stato TODO;
- **Cards_INPROGRESS** -> una lista di supporto per salvare i nomi delle card in stato INPROGRESS;
- **Cards_REVISITED** -> una lista di supporto per salvare i nomi delle card in stato TOBEREVISITED;
- **Cards_DONE** -> una lista di supporto per salvare i nomi delle card in stato DONE;
- **Membri** -> una lista di String contenente i nomi degli utenti che partecipano al progetto;
- **Path** -> una String contenente il path dove salvare/leggere il progetto
- **IP_MULTICAST** -> una String contenente IP per la chat;
- **PORT** -> il numero della porta per la chat.

Al momento che viene chiamato il costruttore della classe, se gli viene passato un path ed il nome di un progetto esistente, il codice cercherà di caricare in memoria tutti i file *.json come card e il file “Users.txt” come lista di utenti che possano accedere al progetto.

Le funzioni create all’interno della classe rispecchiano i comandi che un utente partecipante al progetto può fare:

- AddMember(User) -> permette di aggiungere un nuovo utente alla lista dei Membri;
- AddCard(Name,Desc) -> permette la creazione di una nuova card ed inserirla in stato TODO;
- MoveCard(Name, OldState, NewState) -> permette di muovere la carta corrispondente a Name dal vecchio stato a quello nuovo;
- GetCards() -> restituisce una lista con i nomi delle card all’interno del progetto;
- GetCardHistory(Name) -> restituisce la lista contente la storia dei movimenti della card selezionata;
- GetCardInfo(Name) -> restituisce una lista contente il nome della card corrispondente, la descrizione e lo stato attuale;
- GetMember() -> restituisce una lista contente tutti i membri del progetto;
- Delete() -> equivale ad un endproject, serve ad eliminare il progetto;

Interfaccia WorthServerRMI

WorthServerRMI è un'interfaccia remota che serve al client per chiamare i seguenti metodi sull'oggetto remoto:

- Register(Username,Password) -> permette di registrare un nuovo user, con la relativa password;
- RegisterForCallbacks(User, ClientInterface) -> registra l'oggetto remoto del client per le CallBacks;
- UnregisterForCallbacks(User,ClientInterface) -> elimina l'oggetto remoto del client per le CallBacks.

Classe MulticastIpGenerator

Classe utilizzata per generare una sequenza di IP per il multicast (utilizzati nelle chat), tutti diversi per evitare duplicati. Contiene:

- **IP_lst** -> un insieme di IP tutti diversi;
- **MAX_GEN** -> il numero massimo di IP generati;
- **position** -> il numero di IP gia utilizzati.

Dopo aver chiamato il costruttore il codice genererà direttamente la lista di IP utilizzata. Quindi si potrà chiamare la funzione:

- NextIP() -> rimuove e restituisce un IP dall'insieme di quelli disponibili.

Classe User

La classe User viene utilizzata dal server per salvare le informazioni relative a username e password di ogni utente. Contiene:

- **Username** -> una String relativa al nome utente;
- **Password** -> una String relativa alla password di quell'utente;
- **Online** -> una Boolean che identifica se l'utente è online o no;
- **Nei** -> la relativa interfaccia al client remoto per le CallBacks.

La classe è stata implementata in modo da usare le librerie json per la serializzazione, ad eccezione dello stato online-offline e delle nei, che sono informazioni che andiamo a perdere con lo spegnimento del server, che non ci interessa tenerle sempre salvate.

Classe ServerMain

Classe main del server. Contiene:

- **Progetti** -> la lista dei progetti all'interno del sistema;
- **UPlist** -> lista di utenti salvati nel DB del sistema;
- **Ip_gen** -> istanza di MulticastIpGenerator.

Al momento della chiamata del costruttore, il server cercherà nel DB gli utenti e i progetti e li caricherà in memoria. Le altre funzioni implementate sono quelle per gestire i comandi che i client possono mandare. La comunicazione avviane attraverso la funzione:

- start() -> funzione che permette di leggere i messaggi dai client, gestirli (attraverso execute()) e rispondere agli stessi;
    - Execute(command) -> funzione che viene chiamata internamente a start per la gestione del comando "command". Esegue il comando e genera la risposta.

Essendo ServerMain un'implementazione dell'interfaccia remota WorthServerRMI, avrà anche l'implementazione dei metodi:

- register(user,password) -> funzione chiamata attraverso l'interfaccia remota per registrare un nuovo utente. Dopo la registrazione di un nuovo utente, comunica agli user online l'aggiunta di un nuovo utente (in stato offline), attraverso l'interfaccia per le callabacks;
- registerForCallBacks(nei, user) -> funzione chiamata attraverso l'interfaccia remota per registrare una nuova istanza per le callabacks dell'utente user;
- unregisterForCallBacks(nei,user) -> funzione chiamata attraverso l'interfaccia remota per de-registrare l'istanza per le callbacks dell'utente user.

L'istanziamento del server come oggetto remoto avviane nella funzione main.

Classe MessagingQueue

La classe serve per avere funzioni atomiche sulla lista dei messaggi di una chat. Contiene:

- **queue** -> una lista di String che conterrà tutti i messaggi della chat;

ed i metodi syncronized per operare su di essa:

- put(s) -> inserisce in coda la String s;

- getAndClear() -> restituisce la lista dei messaggi e la resetta (elimina i messaggi già letti).

Classe ChatThread

La classe ChatThread è la vera implementazione della chat fra client del progetto. Contiene:

- **queue** -> una MessagingQueue;
- **PORT** -> porta per la comunicazione della chat;
- **Muticast** -> un multicastsocket per rimanere sempre in ascolto sulla chat;
- **Group** -> il gruppo in cui il multicast si andrà a connettere.

Le funzioni create sono principalmente due:

- sendMsg(msg) -> permette al client di mandare un messaggio nella chat;
- run() -> essendo la classe un estensione di Thread implemento la funzione run in modo che si metta in ascolto nella chat.

Interfaccia NotifyEventInterface

Un'interfaccia remota che serve al server per utilizzare le callbacks, ovvero avvertire il client di cambiamenti, utilizzando le funzioni:

- notifyEventUser(user,state) -> funzione che permette al server di aggiornare lo stato (online-offline) di un utente all'interno delle memoria del client;
- notifyEventChat(IP,PORT,PJT) -> funzione che viene chiamata dal server quando il client viene aggiunto ad un progetto di cui non faceva parte;
- notifuEventProjectCancel(PJT) -> funzione che viene chiamata dal server quando un progetto termina.

Classe ClientMain

Classe principale del client, contiene:

- **UserState** -> una mappa contenente come chiave il nome degli utenti e come valore il loro stato attuale;
- **PJ_Chat** -> una mappa contente come chiave il nome dei progetti e come valore i ChatThread relativi alla chat del progetto;
- **serverRMI** -> oggetto remoto collegato al server;

- **nei** -> istanza remota della stessa classe ClientMain;
- **UserName** -> String contente il nome dell'utente loggato;
- **LOGGED** -> variabile boolean per controllare se l'utente è loggato.

Per quanto riguarda i metodi dopo aver creato un nuovo oggetto ClientMain, passando al costruttore i parametri indirizzo IP e porta del server, si potrà chiamare la funzione start() che sarà la funzione principale del client, che all'inizio andrà a cercare l'oggetto remoto "SERVER". Quindi successivamente, dopo aver aperto la connessione con server tramite SocketChannel, inizierà a leggere i comandi da tastiera attraverso la funzione getNewCommand(), per poi gestirlo attraverso gestCommand(command,client).

Questa funzione va a suddividere tutti i comandi possibili che un client può richiamare e controlla che esso possa effettivamente utilizzarli, attraverso controlli sulla variabile LOGGED ed i comandi CheckPJT, CheckName e CheckUser.

2.2 Strutture dati utilizzate

Durante l'implementazione avevo la possibilità di scegliere diverse strutture dati per la memorizzazione temporanea di informazioni.

Client

All'interno del client ho scelto di utilizzare una Map per l'associazione fra user e il suo stato. Per il Server, invece, ho creato una classe apposita, perché non c'era bisogno di salvare altre informazioni se non il nome dell'utente e il suo stato, Online o Offline: nel caso fosse stato necessario avere altre informazioni sarebbe stato più opportuno avere, come nel server, una struttura a lista di una classe user.

Per quanto riguarda la Map utilizzata per l'implementazione della chat dove ho come chiave i progetti, la scelta di questa implementazione è stata dettata dalla necessità di semplificare la ricerca di una relativa chat: invece di dover scorrere tutta la lista possiamo accedere alla chat relativa al progetto richiesto in tempo costate.

Server

L'implementazione del Server è stata fatta basandosi sulle liste avendo già le classi Project e User che dovevano mantenere molte informazioni: avrei potuto utilizzare le Map per velocizzare la ricerca di un progetto o di un determinato utente ma, in

questo modo, per quanto riguarda gli utenti, ho semplificato l'inserimento in memoria dalla lettura del file *.json.

2.3 Librerie utilizzate

In questo progetto sono state utilizzate librerie già fornite da Java e librerie per l'utilizzo di file *.json, dentro la cartella lib.

3 Threads & Concorrenza

3.1 Lato Server

Il server è formato da un unico thread che gestisce i diversi client connessi attraverso l'uso di un Selector. Funzioni chiamate tramite interfaccia RMI e funzioni per eseguire i comandi chiamati dai client tramite Selector sono state messe come synchronized, in modo da rendere le operazioni atomiche, evitando così corse critiche.

3.2 Lato Client

Il client è leggermente più complesso del server, poiché oltre al thread principale abbiamo in esecuzione contemporaneamente un numero di thread pari al numero di chat di cui l'utente fa parte. Il problema si crea quando si vuole leggere i messaggi da una chat e, allo stesso tempo, se ne ricevono di nuovi dalla stessa chat.

Per ovviare a questo problema è stata utilizzata la classe MessagingQueue, come sistema di memorizzazione della chat, che avendo tutti i metodi syncronized non permette l'accesso in contemporanea di due thread distinti; così facendo le operazioni di lettura e scrittura dall'array formato dai messaggi in chat non avrà problemi di corse critiche.

Visto che il client utilizza anch'esso una interfaccia RMI per modificare gli utenti e i progetti di cui fa parte, è stato necessario implementare due diverse lock, una sui progetti ed una sugli utenti. In questo modo se il client richiede la lista degli utenti online ma allo stesso tempo il server avvisa che un utente si sta disconnettendo, il secondo arrivato dovrà aspettare che la lock sugli utenti sia libera. Stessa cosa per i progetti di cui si fa parte, evitando, in questo modo, il presentarsi di corse critiche.

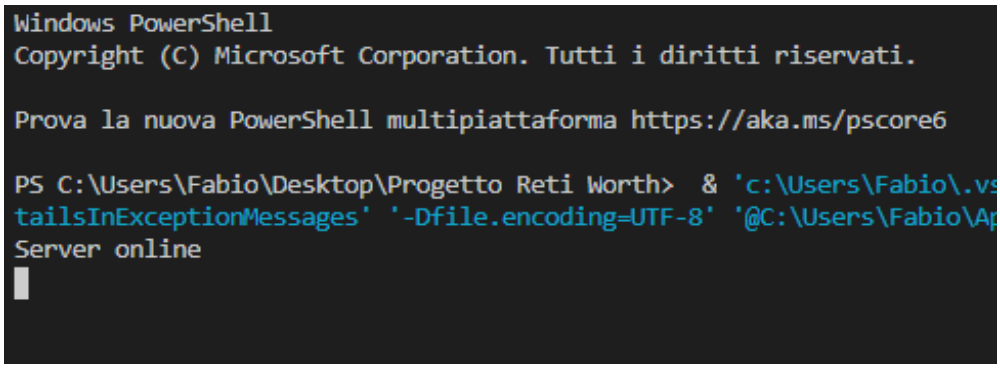
Ci sono alcuni casi però di cui bisogna discutere in modo più approfondito:

- Durante la chiamata di EndProject è ovvio pensare che il client chieda la lock sui progetti, allo stesso modo però anche il server la chiederà quando utilizzerà l'interfaccia RMI per avvisare ogni client della cancellazione di quel progetto, e quindi anche quello che l'ha effettivamente richiesta. In questo modo avremo un problema di deadlock poiché il client non rilascerà la lock fino a quando non riceve la risposta del comando EndProject, e per avere la risposta del comando il server ha bisogno della lock che è ora in possesso del client. Per evitare questo problema è stato fatto in modo che solo i controlli abbiano la lock sui progetti e, quindi, una volta passati i controlli la lock verrà rilasciata ed il server potrà ottenerla senza problemi.
- Nello stesso modo anche la CreateProject soffre dello stesso problema della EndProject, che si risolve nello stesso modo.

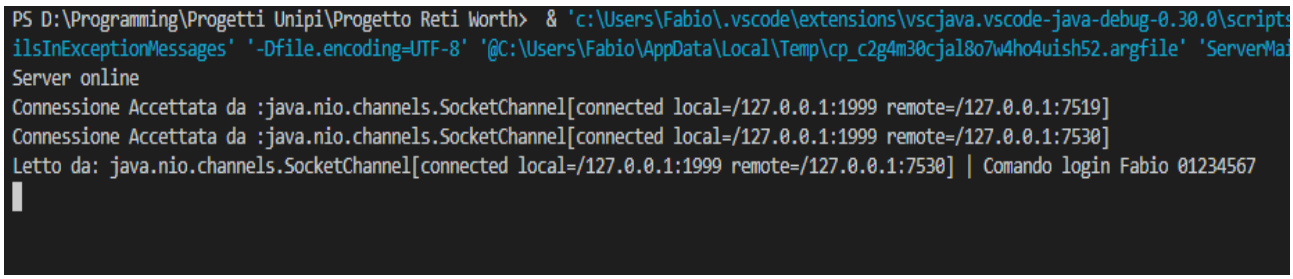
4 Istruzioni per l'uso

4.1 Server

Il server deve essere mandato in esecuzione prima dei client. Una volta mandato in esecuzione verrà stampata la scritta "Server online" se è pronto a ricevere client.



Successivamente il server stamperà un messaggio ogni qualvolta un nuovo client si connette o manda un comando.



4.2 Client

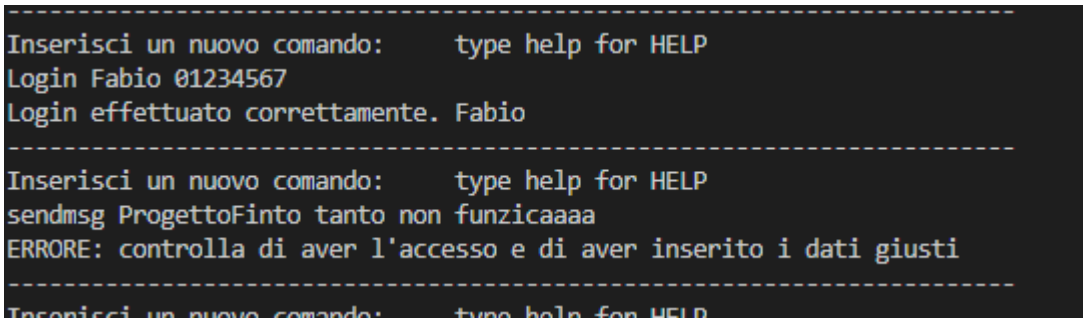
Una volta mandato in esecuzione il server possiamo mandare in esecuzione i diversi client.

Per una lista completa dei comandi che si può utilizzare lato client basta digitare il comando help.

```
------------------------------------------------------------------------
Inserisci un nuovo comando:     type help for HELP
help
--------------------------WELCOME TO HELP DESK----------------------
Ecco una lista dei comandi che puoi usare:
Register 'UserName' 'Password'                                  (**)
Login 'Username' 'Password'                                     (**)
Logout 'Username'
listUser
listUserOnline
CreateProject  'ProjectName'
listProject
addMember 'ProjectName' 'NewUserName'                           (*)
ShowMember 'ProjectName'                                        (*)
ShowCards 'ProjectName'                                         (*)
ShowCard 'ProjectName' 'CardName'                               (*)
AddCard 'ProjectName' 'CardName' 'DESC'                         (*)
MoveCard 'ProjectName' 'CardName' 'OldState' 'NewState'
GetCardHistory 'ProjectName' 'CardName'                         (*)
ReadChat 'ProjectName'                                          (*)
SendMSG 'ProjectName' 'MSG'                                     (*)
EndProject 'ProjectName'                                        (*)
Exit
Help
-----------------------------------NB:-------------------------------
 (**) Utente non deve essere Loggato
 (*)  Utente deve appartenere ai membri di quel progetto
 per tutti i comandi (ad eccesioni di quelli con (**)) devi essere loggato per utilizzarli
------------------------------------------------------------------------
Inserisci un nuovo comando:     type help for HELP
```

Se provassimo a mandare un comando illegale, come ad esempio chiedere l'accesso ad un progetto di cui non si fa parte o utilizzare un comando quando non si è ancora fatto il login, riceveremo un messaggio di errore.

```
Inserisci un nuovo comando:     type help for HELP
sendmsg 02-Prova2 ciao mondo oggi va tutto bene!
User NOT Logged
------------------------------------------------------------
```



Superata la fase di login si ha il libero accesso a tutti i comandi.

4.3 Salvataggio dei dati

Non avendo ancora aggiunto un Admin del server, attualmente ogni client può utilizzare una funzione nascosta chiamata “saveall” che permetterà al server di eseguire il salvataggio di tutti i file presenti al momento del lancio.

5 Conclusioni

5.1 possibili miglioramenti

Il progetto worth non è ancora terminato al 100%: ci sarebbero alcune migliorie da apportare come ad esempio aggiungere un amministratore del server, che permetta di eseguire comandi senza elevati controlli o impostare la possibilità di un salvataggio periodico del server.
Un altro possibile miglioramento è quello di aggiungere un’interfaccia grafica facilitata lato client per la gestione dei progetti, ed una chat migliorata in modo da aggiungere possibilità di conferenze tramite Voip.

5.2 conclusioni

Il progetto WORTH è il progetto del laboratorio di Reti di Calcolatori anno 2021 dell’università di Pisa, dipartimento di informatica.
Si basa sull’utilizzo di tutto quello che abbiamo appreso durante il corso.