# RELAZIONE PROGETTO

# “WORTH”

Francesco Caprari Matricola:580154

## COMUNICAZIONE

La comunicazione tra client e server avviene su una connessione TCP instaurata appena prima che il client effettui il login, il server ha una struttura multithreaded e scrive su una Socket mediante I/O standard.

- LATO CLIENT

  Il client invierà dei messaggi di richiesta ad un producer thread creato dal server scrivendo su un OutputStream collegato al Socket, scriverà l’operazione che il client richiede più una serie di argomenti e si metterà in ascolto su un InputStream aspettando una risorsa o un codice di ritorno che corrisponderà all’esito del comando.

- LATO SERVER

  Il Server per ogni richiesta di connessione avvierà un thread che eseguirà la task della classe producer, all’interno del metodo **run()** vengono letti i comandi grazie all’utilizzo di un InputStream collegato ad una Socket e vengono eseguiti i metodi della classe server corrispondente, il codice di ritorno o la risorsa richiesta vengono scritti sulla socket sfruttando un OutputStream, le eventuali risorse vengono serializzate grazie ad un ObjectMapper.

Oltre alla connessione TCP, i client per gestire i servizi della chat comunicano tra loro sfruttando il protocollo UDP e senza inoltrare direttamente richieste al Server.

# CLASSI

## Card

La classe card rappresenta la card all'interno di un progetto, ci sono gli attributi Name, Descrizione, Lista di appartenenza e un Vector per la History. Utilizzando dei Vector per memorizzare le informazioni delle liste di progetto, quindi delle strutture thrade safe, ho la garanzia di avere delle operazioni atomiche sui dati.

## Progetto

La classe rappresenta un progetto all'interno della piattaforma, ci sono quattro Vector di Card che corrispondono alle quattro liste degli stati possibili e un Vector di String per memorizzare i membri appartenenti al progetto.

Sono presenti i metodi per inserire nuove card nel progetto, per inserire nuovi membri e i metodi necessari per tutte le operazioni di gestione sulle card.

## Server

Si tratta della classe che si occupa in concreto di eseguire le operazioni richieste dal client, all'interno ci sono le strutture dati che memorizzano tutte le informazioni relative ai progetti e agli utenti registrati alla piattaforma.

I metodi che vanno a modificare le liste dei progetti o le coppie utente password sono metodi **synchronized** per avere la mutua esclusione sulla struttura dati, dato il possibile accesso concorrente di più thread e il fatto che si utilizzano delle HashMap che non garantiscono la mutua esclusione.

## MainServer

È la classe dove si definisce il ServerSocket che si occupa di accettare nuove connessioni da parte dei client e in cui si registrano i metodi remoti e dove è presente il CachedThreadpool che manderà in esecuzione i thread(Producer) per la gestione delle connessioni con i client.

## Producer

La classe Producer è la classe che implementa l 'interfaccia Runnable il cui compito è quello di ricevere nello stream le richieste del client e di invocare i metodi della classe server per ottenere i risultati e riscriverli sull' OutputStream per inviarli al client, oltre a mantenere un'istanza della classe server e la socket, mantiene in memoria anche lo username dell'utente per il controllo dei permessi sulle operazioni.

## Client

Ha il compito di comunicare con il server sulla connessione TCP inviando le richieste e ricevendo le risposte sulla connessione instaurata, sono presenti le strutture dati per memorizzare lo stato degli utenti, aggiornate con RMI, ed è presente anche una struttura, aggiornata grazie ad una callback, contenente le coppie <progetto, indirizzo Multicast> utilizzata per mettersi in ascolto sulle chat, è presente anche un Vector per memorizzare le istanze della classe Chat.

## Chat

È la classe che gestisce la lettura e scrittura dei messaggi sulle chat tramite una socket UDP, la classe implementa Runnable e ogni messaggio che riceve dalla socket lo salverà in una struttura dati per leggerlo in futuro.

All'interno della classe ci sono due metodi oltre la run(), il metodo **sendmessage()** per inviare il messaggio sulla DatagramSocket ed il metodo **readmessage()** per leggere i messaggi memorizzati in un Vector che per ogni lettura viene svuotato.

Nel metodo run() il thread del client rimarrà in ascolto finché non viene effettuata una logout.

## InterfaceRegister

L'interfaccia che estende Remote, contiene il metodo **register()** utilizzato dal client per registrarsi alla piattaforma e il metodo **registerForCallback()** utilizzato, una volta effettuato il login, per iscriversi al servizio di notifica degli utenti online.

## InterfaceClient

L'interfaccia estende Remote, al suo interno ci sono i metodi per notificare ai client le informazioni per mandare e ricevere messaggi nelle chat di progetto e per aggiornale la lista di utenti locale.

# Scelte di Progetto

## Persistenza

La persistenza è implementa da diverse classi, la classe server si occupa di serializzare in un formato .json la lista degli utenti della piattaforma nel metodo **register(String nick,String psw)** infatti ogni qual volta si registra un nuovo utente si serializza nuovamente l'HashMap degli utenti, mentre nel metodo **createProject(String ProjectName,String name)** si invoca il metodo **serializ()** (Progetto)sul progetto appena creato che si occuperà di creare la cartella che conterrà i file .json relativi alle card più un ulteriore file per i membri di progetto, quando viene aggiunta o spostata una card viene subito creato o modificato il file .json relativo nei metodi **addCard()** e **moveCard()** della classe Progetto, lo stesso accade quando si aggiunge un nuovo membro al progetto nel metodo **addMember()**

Al primo avvio il Server crea la cartella "Worthsrc" che conterrà tutte le informazioni per la persistenza dei progetti e i file **Register.json** e **MulticastInd.json** rispettivamente per memorizzare le coppie <nomeutente, password> e le coppie <nomeprogetto, Indirizzomulticast>.

All'avvio all'interno del costruttore della classe Server avviene la fase di deserializzazione, inizialmente si ricostruiscono le HashMap per la memorizzazione degli utenti e degli indirizzi multicast dei progetti, in seguito per deserializzare i progetti e le card si scorrono inizialmente tutte le cartelle con i nomi dei progetti e per ognuna di esse si crea un istanza della classe progetto e si invoca il metodo **Restore()** che recupera la lista dei membri e ricostruisce le liste delle card.

Quando un progetto viene eliminato si va anche ad operare sui file eliminando la cartella relativa a quel progetto e si va ad aggiornare anche il file degli indirizzi multicast.

## Aggiornamento lista utenti locale rmi callback

Nel client se il login va a buon fine viene invocato il metodo remoto **registerForCallback()** per registrarsi al servizio di notifica che invocherà il metodo **notifyEvent()** che restituirà le coppie <utente,stato> che andranno ad aggiornare la lista locale del client.

Quindi ogni volta che un utente effettua il login il server scorrerà gli stub memorizzati per aggiornare la lista locale di ogni client.

Anche nel caso del logout, il server si occuperà di rimuove lo stub relativo all'utente ed invocherà nuovamente il metodo **notifyEvent()**.

## Thread e Strutture dati

Il Server avendo una struttura Multithreaded avvierà un thread per ogni client che cerca di connettersi restando in ascolto per ogni richiesta, inoltre nel client, ogni volta che l'utente effettua il login, viene mandato in esecuzione un thread per progetto di cui l'utente è membro per rimanere in ascolto sulle chat.

Nel caso il client faccia il logout il thread non termina, ma rimane in attesa di nuovi comandi, il thread worker termina solamente grazie al comando exit o a causa di una chiusura forzata.

Il Server ha una struttura multithreaded:

- Può servire più client in maniera concorrente
- Il tempo di accettazione della connessione e della response latency è basso finché ci sono un numero ragionevole di client connessi.

Il Server utilizza diverse strutture dati:

**private HashMap<String,String> Utenti; ->** per la gestione delle coppie <utente, password>.

**private HashMap<String,Integer>UtentiStato; ->** per la gestione delle coppie <utente, stato>.

**private HashMap<String,String>ProgettiInd; ->** per la gestione delle coppie <nome progetto, indirizzo multicast>.

**private HashMap<String,InterfaceClient> Stubs;->** per la gestione delle coppie <nome utente, stubs> utilizzato per i metodi remoti.

**private Vector<Progetto> Progetti->** per memorizzare i progetti del server.

A causa dell'architettura del server si deve garantire la mutua esclusione nelle parti di codice dove vengono utilizzate queste strutture.

I vector sono strutture thread safe, mentre tutti i metodi che vanno ad interagire con operazioni di lettura e scrittura sulle HashMap sono metodi **synchronized()** per garantire la mutua esclusione.

## Indirizzi Multicast

Per l'assegnamento degli indirizzi per le chat si sfrutta la funzione **genera()** della classe Server, il metodo viene invocato ogni volta che viene creato un nuovo progetto e sfrutta l'ultimo indirizzo Multicast assegnato per creare il prossimo, il server mantiene le associazioni nome progetto e indirizzo in memoria in maniera persistente.

## Implementazione chat

I servizi della chat sfruttano il protocollo UDP per inviare e ricevere i messaggi, l'utente una volta fatto il login, grazie al meccanismo delle RMI Callbacks, riceverà per ogni progetto di cui fa parte l'indirizzo Multicast su cui mettersi in ascolto, grazie al metodo:

**public void notifyInd(HashMap<String,String> s)**

Quando l'utente richiederà l'invio del messaggio ricaverà la chat relativa al nome di progetto indicato, e invocherà il metodo **sendmessage(String message**) sull'istanza della classe chat.

Inoltre viene utilizzato un ulteriore metodo remotose l'utente viene aggiunto ad un progetto o ne crea uno mentre è già online che invierà solamente una coppia progetto indirizzo.

**public void notifynewInd(String name,String ind)**

Quando un progetto viene eliminato il server invia un messaggio nella chat di gruppo, il client riconoscendo il messaggio chiuderà la chat e farà terminare il thread che era in ascolto.

Il client appena effettuerà il login riceverà dal server le coppie <nomeutente,indirizzo> e chiamerà il metodo **listen()** per mettersi in ascolto sulle chat di ogni progetto.

Nel caso un utente effettui il logout tutti i thread in ascolto sulle chat termineranno.

Anche il server invia messaggi UDP al client sulle chat di progetto per informare i membri del gruppo dell'esito positivo delle operazioni effettuate sulle card o l'aggiunta di un membro al progetto.

## Librerie Esterne

Viene utilizzata la libreria jackson per la serializzazione degli oggetti in un formato json, sia per la scrittura su file, sia per l'invio sulla Socket per restituire le risorse al client.

## Istruzioni di compilazione

Da linea di comando, accedendo alla directory "src" dove sono presenti i file del progetto

*Genera il bytecode i file .class*

- javac -cp ../lib/jackson-core-2.9.7.jar:../lib/jackson-databind-2.9.7.jar:../lib/jackson-annotations-2.9.7.jar MainServer.java Server.java Producer.java InterfaceRegister.java Progetto.java Card.java InterfaceClient.java MainClient.java Chat.java

*Esegue il Server*

- java -cp ../lib/jackson-core-2.9.7.jar:../lib/jackson-databind-2.9.7.jar:../lib/jackson-annotations-2.9.7.jar: MainServer

*Esegue il Client*

- java -cp ../lib/jackson-core-2.9.7.jar:../lib/jackson-databind-2.9.7.jar:../lib/jackson-annotations-2.9.7.jar: MainClient