Relazione Progetto WORTH

1 Panoramica del sistema

Il presente progetto è stato implementato utilizzando la versione Java 11, l'unico modulo di terze parti utilizzato è il pacchetto Gson di Google per le funzioni di serializzazione e deserializzazione dei file .json necessari a garantire il salvataggio dei dati di lavoro. Il programma viene utilizzato mediante interfaccia linea di comando e tutti i comandi disponibili sono riportati all'avvio e richiamabili in qualsiasi momento tramite il comando *help*. Le sincronizzazioni nelle sezioni critiche sono gestite mediante metodi synchronized (nella callback) e blocchi synchronized (nelle messageQueue). Per rendere più leggibile l'interfaccia ho utilizzato le codifiche colori ANSI per colorare parte dei messaggi. Il programma è stato testato su Winsows 7 (locale), Ubuntu 20 (VBox) e Linux Mint Cinnamon (VBox).

2 Esecuzione

Il progetto viene consegnato come archivio compresso contenente la presente relazione e la cartella con i codici sorgente, la sottodirectory *Data*, contenente il database del sistema e 3 script bash .sh per rendere più immediate compilazione ed esecuzione:

- *serverexe.sh*
- *clientexe.sh*
- *clean.sh*

è inoltre fornito il file .jar della libreria Gson di Google per la gestione dei file .json. Per avviare il sitema, posizionarsi con la shell nella directory *Worth* ed eseguire in quest'ordine i file *./serverexe.sh, ./clientexe.sh*. Per ripulire la cartella Worth da tutti i file .class generati dalla compilazione, eseguire lo script *./clean.sh*. In alternativa digitare manualmente i seguenti comandi sul terminale:

- *javac -cp .:./gson-2.8.6.jar *.java* (compilazione file.java e libreria Gson)
- *java -cp .:./gson-2.8.6.jar MainServer* (esecuzione server)

- *java -cp .:./gson-2.8.6.jar ClientApp* (esecuzione client)
- *rm *.java* (per ripulire la cartella)

Ovviamente ogni client dovrà essere eseguito in una nuova sessione di terminale. Per inizializzare da zero il progeramma, azzerando tutti i dati, eliminare la cartella *Data* e cancellare il contenuto del file *lastip.txt.* Nella cartella ho lasciato i dati di alcune esecuizioni di test per mostrare la struttura del database.

3 Classi di appoggio

User: modella un account utilizzatore del sistema, contiene i campi *nickname, password e status(online/offline)* e, oltre ai classici metodi getters e setters comprende *autentication(name,password)* che controlla la corrispondenza delle credenziali passate come parametro, utile per il login.

Card: modella un task di un progetto, un oggetto di tipo Card contiene il suo nome, la descrizione, lo stato di lavorazione (*TODO, INPROGRESS, TOBEREVISED, DONE)* e lo storico degli spostamenti tra una lista e l'altra. Quando viene istanziata una Card viene di default messa nella lista TODO e la voce aggiunta allo storico. I metodi sono i classici getters e setters per accedere ai campi. Per inserire una descrizione composta da più parole è necessario separarle con un "-" (*es. descrizione-della-cardX*).

Project: modella un progetto presente nel sistema, tra le variabili di istanza troviamo le liste per ogni stato di lavorazione contenenti oggetti di tipo Card, una lista di stringhe contenente soltanto i nomi di tutte le card per praticità, un campo con il nome del progetto, un campo con il suo indirizzo multicast, e un oggetto di tipo *File* collegato alla sua directory di salvataggio che viene creata nel momento in cui si istanzia un nuovo oggetto della classe *Project*. Oltre ai classici getters e setters abbiamo il metodo *CopyNremove(list, name)* che serve come appoggio alla funzione *moveCard* quando eliminiamo temporaneamente la card da una lista per inserirla nella nuova, e il metodo *isCompleted* che si occupa di controllare se le liste *TODO, INPROGRESS, TOBEREVISED* siano vuote e quindi il progetto può essere eliminato.

Utils: classe ausiliaria in cui sono definite le costanti di utilizzo comune come i colori ANSI per colorare gli output della console, i path dei file utilizzati per la persistenza, gli indirizzi IP e le porte. Qui è definita anche la funzione *help* che illustra i possibili comandi con relativa sintassi.

4 Persistenza

La persistenza è realizzata mediante una serie di file .json che fungono da "Database", contenuti nella sottodirectory *Data*, più in dettaglio abbiamo:

users.json: contiene l'elenco degli utenti registrati con relativa password e stato.

projects.json: contiene la lista dei dei progetti inseriti a sistema con i relativi campi (liste cards,members,indirizzo).

Data/nomeprogetto: contiene i file .json delle singole cards con tutti i campi dell'oggetto card.

lastip.txt: tiene memorizzato il numero di progetti creati fino a quel momento (compresi quelli eliminati).

All'avvio del server, tramite una chiamata al metodo *loadData()*, viene cercata la cartella *Data/*, se non è presente viene creata, se presente vengono ricaricate tutte le liste leggendo i files al fine di continuare a lavorare con i dati presenti all'ultima interruzione (eccetto i messaggi di chat). I dati vengono scritti ogni qualvolta avviene una modifica (es. aggiunta card, aggiunta progetto ecc...).

5 Client

Il client interagisce con il sistema mediante la classe *ClientApp.java*, qui sono definite tutte le funzioni che permettono di svolgere le varie attività. Per le funzioni di login e logout, sono previti dei controlli per fare in modo che uno stesso utente non possa accedere contemporaneamente da piu client, e non possa disconnettere un accout diverso dal suo. ogni client ha in memoria una coda di messaggi (dettaglio nel paragrafo Chat), una lista con i threads delle chat a cui partecipa e due Hashmap per gestire i servizi di Callback. Il metodo start apre la connessione con i servizi RMI per le funzioni di *register, listusers e listonluneusers*, e la connessione TCP per le restanti funzioni. La funzione start prende in input la stringa di richiesta e mediante opportuni split vengono estratti i comandi e i relativi parametri per poi essere inviati al server tramite la funzione *serverComunica* attraverso SocketChannel. Alla stringa di richiesta viene (in modo trasparente all'utente) concatenato anche il nickname, utile in alcune funzioni per effettuare i controlli sulle autorizzazioni. Le risposte del server vengono poi restitite al client e visualizzate a video dopo essere state eventualmente filtrate. In questa classe è implementata l'interfaccia ClientNotifyINT.

6 Chat

La chat è implementata mediante invio di datagrammi UDP su gruppi multicast. Quando un utente effettua il login, viene avviato un thread (*ChatListenerTH*) per ogni progetto a cui partecipa. Questo thread fa il join sull'indirizzo multicast dei progetti mettendosi così in ascolto sulla porta designata. Quando un messaggio viene captato, questo viene aggiunto ad una lista, ogni utente ha la propria, accessibile al thread di ascolto mediante il metodo *putNewmsg*. Quando

il client invoca il comando *readchat [prog]* vengono mostrati i messaggi da leggere (captati dall'ultima richiesta di lettura) sulla chat del progetto richiesto. E' stato creato un controllo per far si che un utente quando richiede la lettura della chat, non visializzi anche i suoi messaggi. Dopo la lettura i messaggi vengono eliminati dalla coda. Nella coda di un client sono presenti i messaggi relativi a tutti i progetti, vengono poi filtrati in fase di lettura. Per quanto riguarda l'assegnazione degli indirizzi multicast, vengono assegnati mediante una funzione e tenendo memorizzato il conteggio degli indirizzi assegnati in un file (*lastip.txt*) che viene poi utilizzato come offset per le nuove assegnazioni. Non è previsto il riuso degli indirizzi dei progetti eliminati, essendo già possibili inumerevoli indirizzi unici. Per comporre messaggi con più parole occorre separarle da un "-" (Es. *ciao-mario-come-stai*). Gli indirizzi vengono notificati agli utenti tramite callback.

7 Server

Il server è composto da due classi *MainServer.java*, che contiene il metodo main, e la classe *WorthServer.java*. La MainServer non fa altro che creare un'istanza di WorthServer e inizializzare i servizi remoti. Nella WorthServer sono implementate tutte le funzioni necessarie al funzionamento del server, oltre ai metodi dell'interfaccia *RMIServicesINT*.Allinterno della classe abbiamo il metodo *CmdHandler(cmd[])*, che prende in input la stringa del client contenente le istruzioni da eseguire, e dopo aver estratto i parametri (attraverso opportuni split e controlli) esegue le operazioni sulle sue strutture dati in cui sono presenti tutti i progetti, le cards e gli utenti. Naturalmente sono previsti dei controlli per vedere se un client è autorizzato ad eseguire una determinata operazione (es. non posso creare una card in un progetto di cui non faccio parte). Il server comanda anche le scritture dei dati su filesystem ogni qualvolta viene richiesta una modifica. Nel metodo *start(cmd[])* vengono inizializzate le interfacce per la connessione, il server è implementato mediante Multiplexing dei canali (NIO), utilizzando sockets non bloccanti. Questo permette di ridurre i carichi che genererebbe un sistema Multithread che sarebbe costretto a creare un thread per ogni client.

8 Registrazione RMI

Le funzioni di registrazione di un nuovo account è gestita mediante RMI. Nella classe *MainServer* vengono inizializzati il registry e lo stub per esportare l'oggetto. L'interfaccia remota è data dalla classe *RMIServices* e il metodo *register(user, password)* è implementato nella classe *WorthServer*. Quando un nuovo utente si registra viene notificato come "offline" tramite callback (vedi paragrafo successivo)

9 Notifiche Utenti e Indirizzi

Le notifiche sono gestite tramite callbacks, i metodi lato client sono dichiarati nell'interfaccia *ClientNotifyINT* e implementati nella classe *ClientApp* e sono:

- *notifyUsers:* aggiorna la HashMap del client con le informazioni passate dal server.
- *notifySockets:* aggiorna la HashMap del client con le informazioni degli indirizzi multicast dei progetti e crea i threads di ascolto per ricevere i messaggi.

I metodi lato server sono dichiarati nell'interfaccia *RMIServices* e implementati nella classe *WorthServer* e sono:

- *cbSubscribe/cbUnsubscribe*: vengono chiamati dal client per manifestare l'interesse a ricevere gli aggiornamenti.
- *notifyAddress*: chiamato dal server per notificare gli indirizzi multicast che interessano il client (di cui è membro). Crea la HasMap da passare al metodo *notifySockets*.

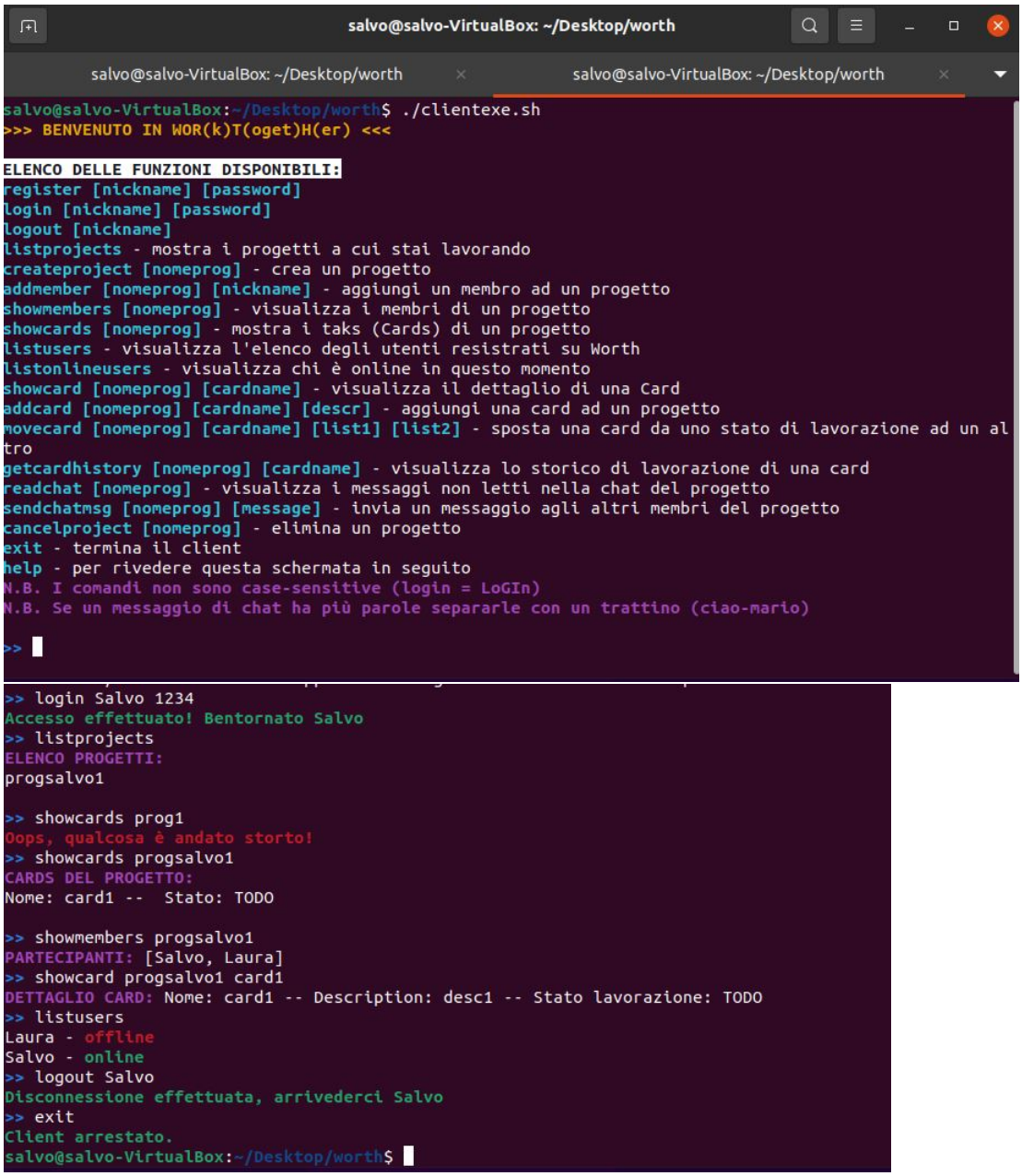

Figure 1: Esempi del client in funzione.