UNIVERSITÀ DI PISA

Relazione progetto WORTH v.1.1

Laboratorio di Reti di Calcolatori A
Prof.ssa L.M.E. Ricci
A.A. 2020/2021

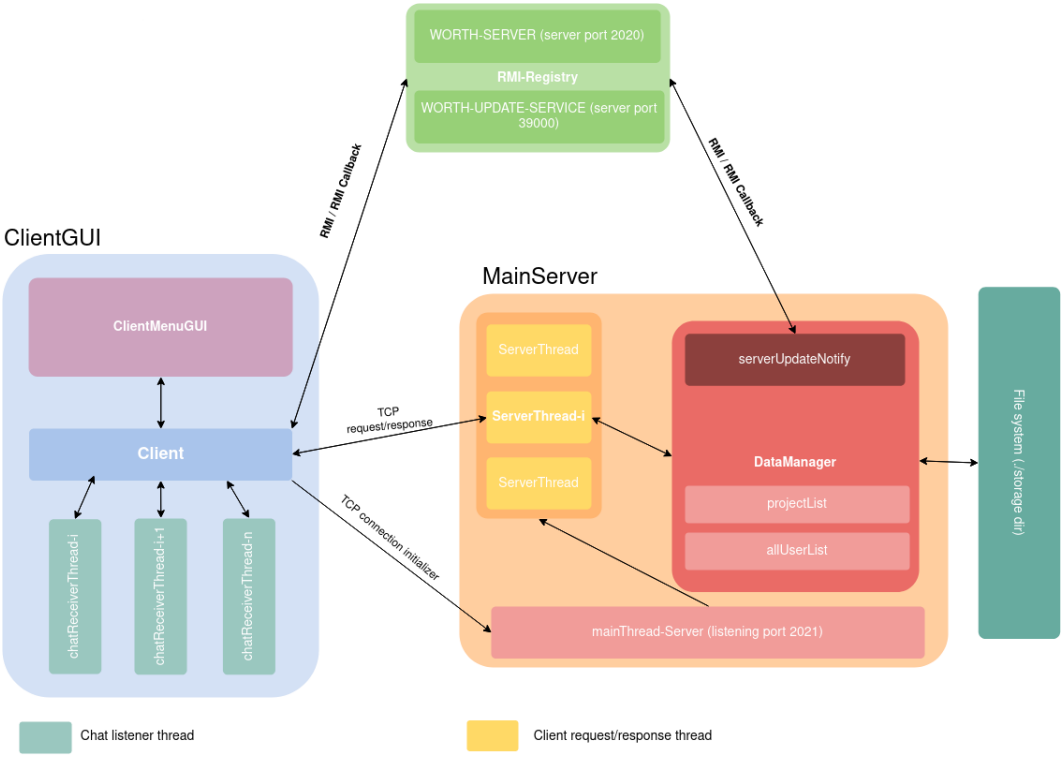
Luca Giovambattista Pinta
Matricola 579458

January 01, 2021

Contents

1 Architettura di sistema

Di seguito uno schema semplificato dell'architettura di sistema e delle relative interazioni con componenti e relativi threads.



L'implementazione è articolata in due classi principali , **ClientGUI** e **MainServer** contenenti i rispettivi metodi *main* di avvio di client e server. Nello specifico:

1.1 ClientGUI

Il client è dotato di interfaccia grafica implementata tramite libreria *JavaSwing*. L'interfaccia grafica è implementata nella classe **ClientGUI** (*contenente schermate di login, registrazione, finestra principale, finestra chat*) e nella classe **ClientMenuGUI** (*contenete la dashboard di gestione del progetto selezionato*). La componente grafica interagisce con la classe **Client** che si occupa della gestione dati lato client e di effettuare richieste al server e di restituire una risposta in un formato consono alla visualizzazione grafica.
Il formato delle richieste da client a server è della forma:

$$ACTION - REQUEST\#parameter1\#paramter2\#...\#parameterN \quad (1)$$

in cui *ACTION* indica la rappresentazione in formato stringa dell'azione richiesta al server e i *parameters* i relativi parametri di ingresso, se presenti. Dualmente, il formato delle risposte da parte del server è della forma:

$$data1\#data2\#...\#dataN \tag{2}$$

in cui se *ACTION* prevede solo una risposta di conferma o la restituzione di una notifica di errore sará indicato da una stringa (*OK, ERROR, etc. Si veda* ***Codex*** *per la lista estesa delle richieste/risposte*). Il carattere # verrá considerato come carattere nullo e dunque non accettato in input per l'immissione di informazioni sul progetto e delle relative cards.
Poichè il client dotato di interfaccia grafica fornirá all'utente alcune funzionalitá base componendo piú funzionalitá descritte nella specifica di progetto (*un esempio è la visualizzazione chat progetto: il client prima richiederá al server la lista dei progetti aggiornata, visualizzandola graficamente, e infine visualizzerá la chat di uno specifico progetto selezionato*). Lo stato dettagliato di richieste/risposte e delle singole azioni eseguite dal client è visibile da terminale.
Relativamente alla gestione della chat, il client dopo aver effettuato il login entrerá a far parte dei gruppi multicast dei progetti a cui appartiene (*poichè avrá ricevuto dal ServerThread la lista dei progetti e dei rispettivi indirizzi IP multicast a cui unirsi*) e per ognuno di essi verrá attivato un **chatReceiverThread** che funge da *listener* al fine di permettere la visualizzazione dei messaggi inviati in chat da altri membri del progetto in un secondo momento, mantenendo una *cronologia* della chat. Nel caso in cui un utente venga aggiunto ad un progetto in un secondo momento rispetto al login non verrá attivato alcun *thread listener* fino al momento in cui il client non entrerá esplicitamente (*tramite interazione GUI*) per la prima volta nella chat del progetto a cui è stato aggiunto.
Si è scelto di seguire un **pattern di gestione degli errori guidato dalle eccezioni**: nello specifico, tale pattern è evidente nel caso di una chiusura inaspettata del server provocando il sollevamento di una *NullPointerException* che consente di avvisare il client dell'evento e di terminare dunque l'esecuzione del client stesso.
Cosí come da specifica progettuale, il meccanismo RMI è utilizzato sia per la registrazione che per la notifica di variazioni dello stato di connessione di un utente: il client effettuerá una lookup sull'RMI registry rispettivamente degli oggetti **WORTH-SERVER** e **WORTH-UPDATE-SERVICE**.
Si è scelto inoltre l'utilizzo di segnali (*sun.misc.Signal*) al fine di permettere una corretta esecuzione delle operazioni di shutdown alla ricezione del segnale di **SIGINT**, in caso di interruzione improvvisa, sia lato client (*chiusura connessione TCP, leaving gruppi multicast, interruzione thread listener*) che lato server (*interruzione ServerThread, trattata nella sottosezione seguente*). L'operazione di logout è stata interpretata come una chiusura corretta da parte del client, dunque causa la chiusura del client.

1.2 MainServer

Il server implementa le funzionalitá descritte dalla specifica progettuale tramite cinque classi principali (*escludendo la classe **Card** di cui omettiamo la trattazione non avendo alcuna rilevanza particolare se non quella che si evince dalla specifica progettuale*), ognuna con un ruolo definito:

- **MainServer**: si occupa di accettare le connessioni in entrata (*sulla socket TCP*) e di associare un **ServerThread** per ogni client connesso al server, delegando la gestione delle richieste di un singolo client al rispettivo thread server, eseguito come task da un **newCachedThreadPool**. Gestisce lo startup dell'RMI registry e dell'esportazione degli oggetti remoti deputati alla registrazione (***WORTH-SERVER**, porta 2020*) e alla gestione delle notifiche utente tramite callback RMI (***WORTH-UPDATE-SERVICE**, porta 39000*) implementando inoltre il metodo *register* dell'oggetto remoto messo a disposizione del client per la registrazione.
  Si è scelto, analogamente al Client, l'utilizzo di segnali (*sun.misc.Signal*) al fine di permettere una corretta esecuzione delle operazioni di shutdown alla ricezione del segnale di **SIGINT**, implementando la permanenza dei dati su file system (*inerenti le informazioni dei progetti, mentre la permanenza dei dati utente è implementata all'atto della registrazione*).

- **ServerThread**: gestisce le richieste in entrata da parte del client associato e implementa la funzionalitá di avviso nella chat di progetto in caso di modifica della lista di appartenenza di una card (*se il client associato effettua una modifica sullo stato di una card il thread effettua una join nel gruppo multicast del progetto specificato inviando un messaggio di aggiornamento card, effettuando infine una leave del gruppo multicast*).
  La classe *ServerThread* implementa la ricezione delle richieste secondo la notazione descritta precedentemente in *(1)* e implementando l'esecuzione delle azioni richieste dal client tramite il supporto della classe *DataManager*. Alla ricezione della richiesta di logout da parte del client associato il thread terminerá la propria esecuzione, chiudendo la connessione TCP.

- **DataManager**:a supporto della classe *ServerThread*, implementa le azioni richieste dai singoli client sulle strutture dati condivise oltre che la gestione delle strutture dati per i client, espletando le funzioni di arbitro della concorrenza. Tale classe contiene l'implementazione effettiva delle operazioni descritte nella specifica progettuale (*sia in modo diretto tramite metodi implementati nella classe stessa che indiretto tramite l'uso di metodi offerti dalla classe di supporto **Project***). Gestisce il caricamento dei dati all'avvio del server e del salvataggio/aggiornamento dei dati su file system alla chiusura del server: la **permanenza** dei dati è assicurata dalla scrittura sul *file system* tramite la serializzazione della lista utenti nel file **users.json** e per i progetti (*contenuti nella main folder /projects/*) rispettivamente un file *.json* per ogni **Card** e un file generale di progetto

project.txt contenente le informazioni secondo la notazione:

$$projectName\#[member1, member2, ..., memberN]\#multicastIP \quad (3)$$

che consentono di ricostruire lo stato di un progetto.Tale classe gestisce inoltre l'invocazione dei metodi addetti alla notifica di variazione dello status degli utenti tramite il supporto della classe *ServerUpdateNotify*.

- **Project**: contiene metodi specifici utilizzati a supporto della classe *DataManager* inerenti la modifica, creazione ed eliminazione di un singolo progetto tramite il supporto della classe *Card*. Contiene inoltre dei metodi ad uso esclusivo dei metodi definiti per effettuare ripristino/scrittura dei dati su *file system* (*definiti nella classe DataManager*).

- **ServerUpdateNotify**: a supporto della classe *DataManager*, implementa le funzionalitá di registrazione/deregistrazione dal servizio di notifica RMI callback per la variazione dello status degli utenti e contiene l'effettiva implementazione del metodo *update(username,status)* deputato alla notifica degli eventi ai client registrati per il servizio.

2 Strutture dati e sincronizzazione

Analizziamo rispettivamente per client e server le strutture dati utilizzate e i relativi meccanismi di sincronizzazione sulle strutture dati critiche:

2.1 Client

Le principali strutture dati utilizzate sono quattro differenti *HashMap* allocate dalla classe *Client* cosí definite:

- **HashMap⟨String,String⟩ userList**: contiene la coppia ⟨username,passowrd⟩ che rappresentano rispettivamente la coppia ⟨chiave,valore⟩. Tale struttura dati rappresenta la lista utenti iniziale (*aggiornata da successive callback RMI*), è allocata nel metodo costruttore di classe e popolata in caso di esitivo positivo al *login* con i dati mandati dal rispettivo *ServerThread*.

- **HashMap⟨String,MulticastSocket⟩ multicastRegister**: contiene la coppia ⟨multicastIP,groupSocket⟩ che identificano univocamente la socket del gruppo multicast con indirizzo *multicastIP*. Tale struttura dati è utilizzata per tener traccia dei gruppi multicast dei rispettivi progetti e poter effettuare le azioni di *leave/join* negli opportuni casi.

- **HashMap⟨String,ChatReceiverThread⟩ clientListener**: contiene la coppia ⟨projectName,ChatReceiverThread⟩ che identifica univocamente il thread listener in ascolto sulla chat del progetto *projectName* al fine di mantenere una history della chat e di renderla visibile tramite GUI.

- **HashMap⟨String,String⟩ chatHistory**: contiene la coppia ⟨projectName,history⟩ che identifica la cronologia della chat di *projectName*. Tale struttura è utilizzata sia dai *ChatReceiverThread* (*thread listener*) sia dalla classe *ClientGUI* per la corretta visualizzazione della cronologia nella componente grafica (*JTextArea in JavaSwing*).

Le strutture dati lato client non richiedono alcun meccanismo di sincronizzazione poichè eventuali azioni concorrenti sono arbitrate dal server.

2.2 Server

Le principali strutture dati utilizzate sono quattro differenti *HashMap* allocate dalla classe *DataManager* e sei differenti *ArrayList* allocati nelle classi *Project* e *Card*, cosí definite:

- **HashMap⟨String,String⟩ allUserList**: contiene la coppia ⟨username,password⟩ che rappresentano rispettivamente la coppia ⟨chiave,valore⟩. La sincronizzazione su tale struttura dati è implementata tramite l'uso di blocchi *synchronized* e arbitrata dalla classe *DataManager* e mediante l'uso di *metodi synchronized* contenuti nella classe *MainServer*.

- **HashMap⟨String,MulticastSocket⟩ allUserListStatus**: contiene la coppia ⟨username,status⟩ dove status puó assumere i valori *online o offline* e rappresenta lo stato degli utenti noto al server. La sincronizzazione è implementata mediante l'uso di blocchi *synchronized* sulla struttura dati e arbitrata dalla classe *DataManager*.

- **HashMap⟨String,Project⟩ projectList**: contiene la coppia ⟨projectName,Project⟩ che identifica univocamente un progetto. Nonostante la ridondanza dell'informazione *projectName* (*contenuto sia come chiave nella struttura sia nell'oggetto Project come attributo*) si è scelta tale struttura per l'ottimalitá nel costo di ricerca. La sincronizzazione è implementata mediante l'uso di blocchi *synchronized* sulla struttura dati e arbitrata dalla classe *DataManager*.

- **HashMap⟨String,UserUpdateNotify⟩ clients**: contiene la coppia ⟨username,clientInterface⟩ utilizzata per la registrazione/deregistrazione e la relativa calllback di aggiornamento dell'utente *username*. La sincronizzazione è implementata mediante l'uso di *metodi synchronized* implementati nella classe *ServerUpdateNotify* e tramite l'uso di blocchi *synchronized* sulla struttura dati all'interno della classe *DataManager*.

- **ArrayList⟨String⟩ userList**: contiene la lista membri del progetto in formato *String*. La sincronizzazione è implementata mediante l'uso di blocchi *synchronized* sulla struttura dati e arbitrata dalla classe *DataManager*.

- **ArrayList⟨Card⟩ todo/inProgress/toBeRevised/doneList**: rappresentano i quattro stati in cui puó trovarsi una *Card*. La sincronizzazione

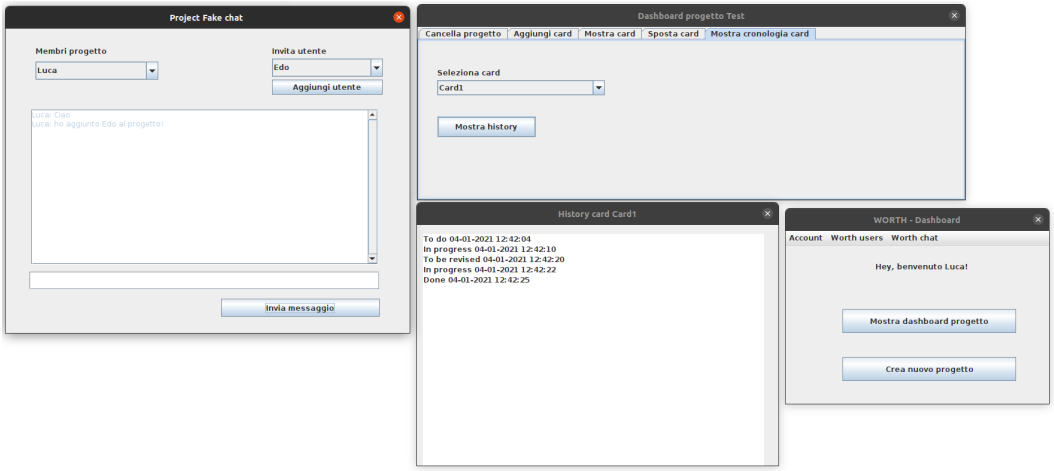
è implementata tramite l'uso di blocchi *synchronized* contenuti nella classe *DataManager*.

- **ArrayList⟨String⟩ history**: contiene la cronologia delle variazioni di una specifica Card nel formato stringa secondo la notazione *stato - dd-MM-yyyy HH:mm:ss*. è contenuta nella classe *Card* che delega la sincronizzazione della struttura dati alla classe *DataManager*.

Si è scelto di identificare nella classe *DataManager* un **arbitro della concorrenza**: seppur tale scelta possa sembrare un *collo di bottiglia* in termini di prestazioni sulle operazioni richieste dai client, l'utilizzo diffuso di *HashMap* consente un netto miglioramento delle prestazioni seppur a discapito di una granularitá dei meccanismi di locking meno specifica. Vista la natura delle funzionalitá offerte dal server ed effettuata una valutazione di strutture dati in grado di garantire una granularitá minore dei meccanismi di locking, si è scelto di adottare largamente l'uso di *HashMap* in combinazione all´uso di blocchi *synchronized* preferendo un incremento delle prestazioni sulle singole operazioni piúttosto che l'adozione di meccanismi di locking a grana fine.

3 Interazione GUI

Si è fatto utilizzo della libreria di Java **JavaSwing**: la gestione delle schermate è determinata dalle risposte ottenute dalla classe *Client* che permette la restituzione di eventuali dati in forma consona alla visualizzazione grafica. Per la gestione delle schermate si fa uso delle componenti *JFrame,JDialog*.



Nonostante si sia fatto uso della libreria *JavaSwing* non è stato utilizzato il supporto per l'aggiornamento asincrono delle componenti d'interfaccia operato da **SwingWorkers**: l'aggiornamento è dunque operato principalmente effettuando un *refresh* delle informazioni (*generalmente operato tramite JComboBox*

di selezione, in modo trasparente all'utente).
Pur rispettando le specifica progettuale, l'integreazione dell'interfaccia grafica ha reso di dubbia utilitá l'implementazione di due metodi specifici contenuti nella specifica progettuale nonostante tali funzionalitá siano state effettivamente implementate: l'implementazione di un metodo **readChat** in presenza dell'interfaccia grafica risulta superfluo poichè tale azione è demandata principalmente alle componenti grafiche di *JavaSwing* utilizzate e al relativo **thread listener**, di fatto non necessitando alcuna interazione con il nucleo delle operazioni svolte lato client dalla classe *Client*. Dualmente, la forte integrazione con l'interfaccia grafica ha reso superfluo l'implementazione del metodo **sendChatMsg** poichè tale azione è operata dall'interazione con la struttura dati *multicastRegister* contenuta nella classe *Client* e dalle componenti grafiche di *JavaSwing*.
Un maggiore grado di disaccoppiamento tra il *core* di funzioni svolte dal client e le relative componenti grafiche avrebbe consentito una maggiore riusabilitá del codice e una struttura adatta a sviluppi successivi.

4 Compilazione e dependencies

Il progetto fa uso della libreria esterna FasterXML Jackson ed è stato sviluppato utilizzando Java 11 e lo strumento di gestione dei progetti **Maven** alla versione *3.6.3*.
Per la compilazione è necessario aggiungere i file *.JAR* di **Jackson** *databind,core e annotations* al *CLASSPATH* del progetto (*qui riferimenti dettagliati)* ed eseguire la compilazione in java indicando tramite il flag *-cp* il CLASSPATH.
In alternativa, tramite **Maven** è possibile compilare ed eseguire tramite le *dependencies* riportate nel file *pom.xml*:

Listing 1: pom.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
                apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>org.example</groupId>
8      <artifactId>P1</artifactId>
9      <version>1.0-SNAPSHOT</version>
10     <build>
11         <plugins>
12             <plugin>
13                 <groupId>org.apache.maven.plugins</groupId>
14                 <artifactId>maven-compiler-plugin</artifactId>
15                 <configuration>
16                     <source>8</source>
17                     <target>8</target>
18                 </configuration>
19             </plugin>
20         </plugins>
```

```
    </build>
    <dependencies>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
            <version>2.9.6</version>
        </dependency>

        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
            <version>2.9.6</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.9.6</version>
        </dependency>

    </dependencies>

</project>
```

Per **compilare** il progetto con le dipendenze Maven, tramite prompt dei comandi spostarsi nella directory contenente il file *pom.xml* ed eseguire:

mvn compile

Per **eseguire** il **server** tramite prompt dei comandi spostarsi nella directory contenente il file *pom.xml* ed eseguire:

mvn exec:java -Dexec.mainClass=MainServer

Rispettivamente, per il **client**:

mvn exec:java -Dexec.mainClass=ClientGUI

Sia client che server non richiedono parametri d'ingresso per l'esecuzione.

5 Codex

Si riporta la lista dei metodi e la relativa codifica utilizzata per richiedere al server tramite *socket TCP* l'esecuzione dell'operazione corrispondente e i relativi stati di errore/dati restituiti (*in verde in caso di esito positivo, in rosso in caso di esito negativo*).

Metodo client	Codifica	Server response
login(username,password)	LOGIN-REQUEST#username#password	#username#status USERNAME-ERROR PASSWORD-ERROR ONLINE-ERROR
logout(username,logged)	LOGOUT-REQUEST#username	OK ERROR
listProjects(username,multicast)	LIST-PROJECT-REQUEST#username	#project#multicastIP# ERROR
createProject(projectName)	CREATE-PROJECT-REQUEST#projectName	OK PROJECT-TITLE-ERROR
addMember(project,member)	ADD-MEMBER-REQUEST#project#member	OK ALREADY-MEMBER ERROR
showMembers(project)	SHOW-PROJECT-MEMBERS#project	#username# ERROR
showCards(project)	SHOW-CARDS-REQUEST#project	#cardName# NO-CARDS-ERROR
showCard(project,cardName)	SHOW-CARD-REQUEST#project#cardName	#stato#descrizione# CARD-ERROR
addCard(project,cardName,desc)	ADD-CARD-REQUEST#project#cardName#desc	OK PROJECT-ELIMINATED CARD-DUPLICATE-ERROR
moveCard(project,card,destList)	MOVE-CARD-REQUEST#project#card#destList	OK COINCIDENT-LIST STATE-VIOLATION INTERNAL-ERROR PROJECT-ELIMINATED
cancelProject(projectName)	CANCEL-PROJECT-REQUEST#projectName	OK CARD-STATUS-ERROR PROJECT-ELIMINATED