Università di Pisa

Laurea Triennale in Informatica
Corso di Laboratorio di Reti

Relazione del progetto

WORTH

Docente:
Federica Paganelli

Studente:
Davide Chen | 544795

Anno Accademico 2020/2021

Indice

1 Introduzione

Il progetto è stato realizzato utilizzando l'ambiente di sviluppo IntelliJ IDEA con JAVA JDK versione 12.0.2 su sistema operativo Windows 10. Segue le specifiche date e fa ausilio delle librerie Jackson (versione 2.9.7) per la serializzazione e de-serializzazione. L'applicativo utilizza un'interfaccia a linea di comando e fa utilizzo delle ANSI Escape Sequences per la differenziazione dei colori per i vari tipi di messaggi al fine di avere un interazione più user-friendly.

2 Descrizione dell'architettura

Il programma segue il paradigma client-server, per cui si compone di 2 parti:

- componente server
- componente client

2.1 Comunicazione

I componenti comunicano utilizzando il meccanismo RMI per gli eventi di registrazione e RMI Callbacks per quelli di aggiornamento, mentre per tutte le altre operazioni usano una connessione TCP, il quale è persistente per tutta la durata della comunicazione. L'indirizzo e porta della connessione sono note e dichiarate nella classe astratta CommunicationProtocol, dove oltre a essere presente le informazioni per localizzare il registry per ricavare gli oggetti remoti pubblicati dal server, troviamo tutte le informazioni utili per il corretto funzionamento del sistema come la lista dei comandi e i statusCode che possono essere prodotti.

Attraverso la connessione TCP instaurata, client e server utilizzano un protocollo di comunicazione per lo scambio di messaggi con la seguente struttura:

Messaggio di richiesta (RequestMessage)

- Command: operazione che il client richiede al server
- Arguments: sequenza degli argomenti che il client fornisce al server per elaborare la risposta

Messaggio di risposta (RespondeMessage)

- Status code: codice di risposta alla richiesta
- Response body: risorsa richiesta dal client (opzionale)
- Response body 2: questo campo è utilizzato solo dal comando *login()*, per ricevere la mappa contenente gli indirizzi di chat Multicast

I messaggi scambiati fra il client e il server sono codificati come stringhe in formato JSON.

A seguito della registrazione dell'utente, il client potrà effettuare l'operazione di login, l'unica operazione che permetterà di inviare ulteriori richieste.

Quando l'utente effettua il login, il server include nel messaggio di risposta la lista degli utenti e i loro rispettivi stati (responseBody) + la lista degli indirizzi IP Multicast (responseBody2) per attivazione del Thread per la ricezione dei messaggi di chat. Dopodiché si registra al servizio di Callback e da lì in poi, l'utente verrà aggiornato in modo asincrono dal server sui cambiamenti di stato degli altri utenti.

Il server, in base al comando ricevuto, proverà ad elaborare una risposta per il client. Questa risposta sarà comprensiva di status code (se la richiesta è terminata con successo o, in alternativa, l'errore riscontrato). Il Response Body, ove presente, è una risorsa strutturata e con un formato concordato da cliente e server: il client sa che dovrà ricevere una lista piuttosto che una struttura map.

Lo schema di funzionamento è il seguente:

1. L'utente immette il comando richiesto
2. ClientService prepara messaggio di richiesta, lo serializza e lo invia al server tramite la socket TCP
3. Server riceve il messaggio, lo de-serializza ottenendo un RequestMessage
4. Analizza il comando e argomenti
5. Chiama funzione interna di Database
   (a) Se l'operazione viene effettuata, il server preparerà un messaggio di risposta ResponseMessage con statusCode = SUCCESS (codice 200) e, se richiesto dal comando di richiesta, serializzerà il risultato ottenuto dalla funzione primitiva e lo inserirà in responseBody
   (b) Se l'operazione lancia un eccezione X, questa viene catturata e viene generato uno statusCode = VALORE-ECCEZIONE-X
6. Server serializza il messaggio di risposta e lo invia al client tramite la socket TCP
7. ClientService riceve risposta, la de-serializza ottenendo un ResponseMessage
   (a) Se statusCode == SUCCESS vuol dire che l'operazione è andata a buon fine
   (b) Se statusCode == VALORE-ECCEZIONE-X allora il client lancia l'eccezione X
8. Questa eccezione viene catturata dal ClientMain e stampa sul terminale un messaggio di errore

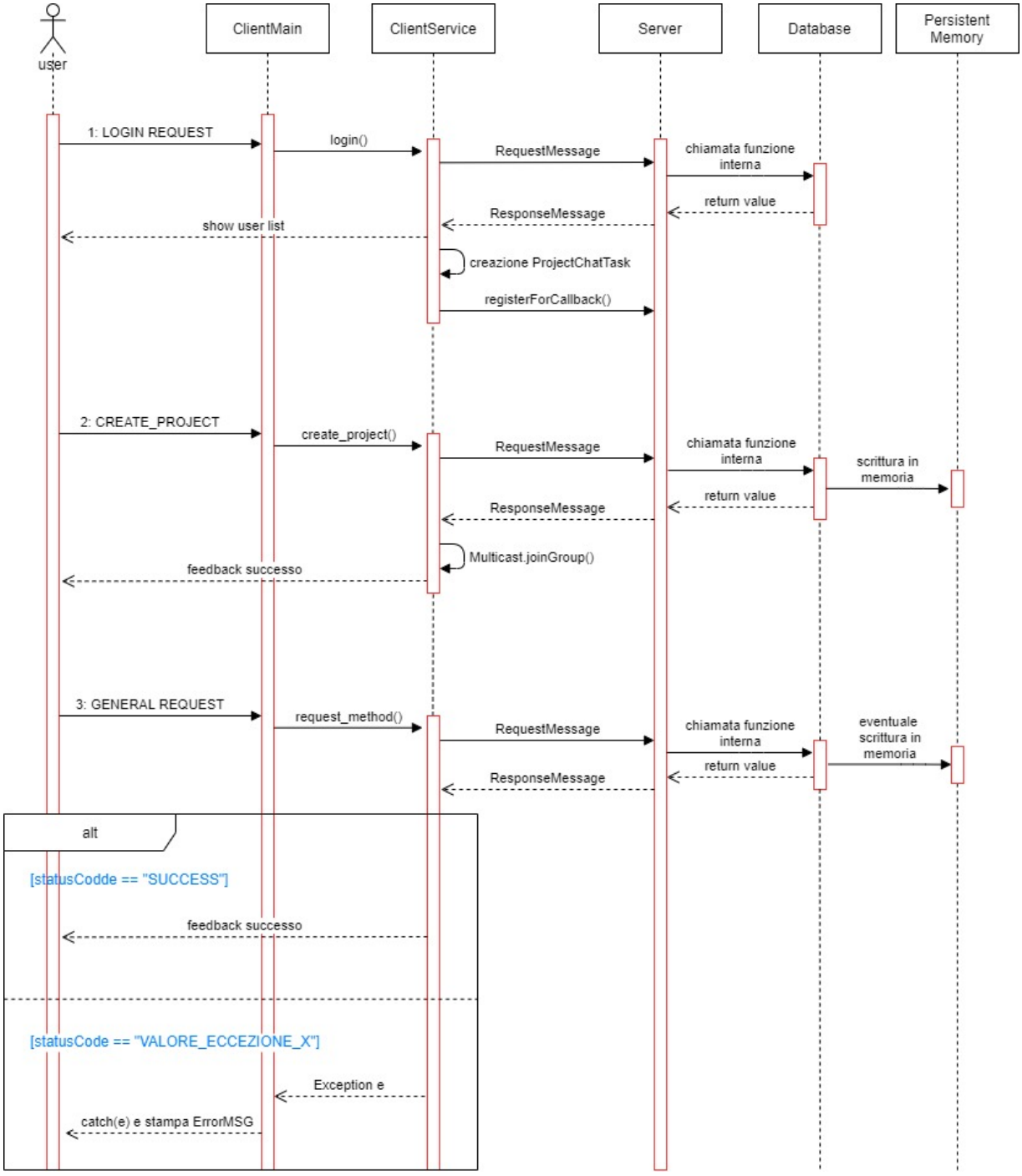

Figura 1: Diagramma di sequenza che mostra un esempio di interazione del sistema

La gestione delle connessioni è effettuata tramite Multiplexing dei canali mediante NIO e nel caso dei metodi RMI, la concorrenza è gestita tramite l'acquisizione delle lock implicite mediante la keyword "synchronized".

Oltre al TCP, nel progetto vi è anche l'utilizzo del protocollo UDP per il servizio di chat, il quale è interamente gestito dal client, tranne che per l'assegnazione degli indirizzi IP Multicast, i quali vanno richiesti al server, mentre la porta è standard per tutti.

2.2 RMI per le operazioni di Registrazione e Callback

Come già accennato sopra, le operazioni di registrazione di nuovi utenti al servizio e callback di notifica del cambio stato dei utenti sono state implementate con il protocollo RMI. In aggiunta a queste operazioni, è stato fatto uso del meccanismo di RMI Callback anche per le seguenti operazione:

- l'aggiornamento delle informazioni delle connessioni Multicast, dove a seguito dell'operazione *addMember()*, il nuovo utente aggiunto viene notificato con i dati necessari per attivare la nuova chat di progetto.
- l'abbandono di un gruppo Multicast a seguito della cancellazione di un progetto
- notifica dell'arresto del server per invitare gli utenti a chiudere i proprio applicativo client

2.3 Funzionamento chat di progetto

La chat di progetto è implementata utilizzando IP Multicast, il quale permette ad un singolo client di inviare un datagramma UDP ad un insieme di client che si erano preventivamente uniti al gruppo, e i messaggi di chat possono essere inviati direttamente da client a client, senza passare per un server intermediario. Le funzionalità della chat sono state implementate nella classe ProjectChatTask. Al momento dell'istanziamento dell'oggetto, sono inizializzate le strutture dati per la memorizzazione dei riferimenti e dei messaggi ricevuti durante la sessione, ed è aperto un SOLO socket UDP per la ricezione dei pacchetti Multicast.

Lo schema dell'attivazione e funzionamento della chat è il seguente:

1. L'utente fa il login con il quale riceve dal server una mappa, contenente per ogni entry la coppia $<projectName, chatAddress>$
2. Si istanza un oggetto della classe ProjectChatTask e viene passato la mappa ricevuta come argomento
3. La task viene passato ad un Thread che una volta attivato fa una joingGroup sulla multicastSocket per ogni entry della mappa, istanza una DatagramPacket e si mette in ascolto per ricevere nuovi messaggi
4. Quando l'utente invoca il comando readChat(), il programma fa restituire dalla task gli ultimi messaggi ricevuti e non ancora letti per poi stamparlo sulla console.
5. Mentre per il comando sendChat() fa restituire l'indirizzo IP e multicastSocket, per preparare il DatagramPacket per poi spedirlo
6. Ulteriori nuovi gruppi di chat possono essere attivati alla creazione di nuovi progetti o quando si è stato aggiungo ad un nuovo progetto esistente, in quest'ultimo caso viene gestito tramite il metodo RMI Callback, il quale notifica, all'utente interessato, i dati necessari.
7. In caso il progetto, a cui la chat fa riferimento, viene cancellato, il server manda un avviso di sistema sulla chat e poi procede a far abbandonare il gruppo Multicast a tutti i membri, sempre con il meccanismo di RMI Callback.
8. Al logout dell'utente viene invocato il metodo terminate(), il quale non fa altro che terminare il thread e fare la leaveGroup su tutti i gruppi a cui si è uniti.

9. Quando un progetto viene cancellato, l'indirizzo viene liberato per essere riutilizzati in un nuovo progetto.
10. In base alla tipologia di messaggio ricevuto, questi vengono mostrati sul terminale con colori differenti:
    - giallo per i messaggi di sistema
    - blu per messaggi inviati da me (cioè i messaggi inviati dall'utente loggato su quel client)
    - bianco per i messaggi inviati da tutti gli altri utenti

```
Insert next command:
> read_chat prova2
---------- Reading chat ------------
System: usr1 moved card 'card3' from TODO to INPROGRESS
System: usr1 moved card 'card2' from TODO to INPROGRESS
usr2: ciao bello
usr1: heyla come stai carissimp
System: usr1 moved card 'card3' from INPROGRESS to TOBEREVISED
usr2: come stai ultimamente??
usr2: che fai di bello?
usr1: tutto bene nulla di nuovo
System: usr1 moved card 'card3' from TOBEREVISED to DONE
------------ End chat --------------
```

Figura 2: Esempio di interfaccia chat a seguito della chiamata a *readChat()*

2.4 Server

Il server è il fulcro del servizio WORTH: si occupa di gestire le connessioni di rete verso i client e mantiene al suo interno le strutture dati necessarie al soddisfacimento delle richieste; si occupa di salvare in memoria secondaria i dati relativi allo stato della piattaforma, nonché del ripristino delle strutture in seguito al riavvio. L'implementazione delle funzionalità di rete e la gestione delle connessioni è racchiusa interamente nella classe **SelectionTask**, mentre le operazioni di gestione delle operazioni e salvataggio e ripristino dei dati in memoria permanente è delegato alla classe **Database**.

Il server è implementato mediante lo schema del **NIO Selector** e **SocketChannel non bloccanti** per la gestione delle connessioni. Questa scelta è stata effettuata poiché:

1. È una soluzione che scala in maniera efficiente quando in numero di connessioni instaurate cresce
2. Garantisce un efficiente utilizzo delle risorse
3. Garantisce la sequenzialità delle operazioni e consistenza dei dati, evitando così la gestione della concorrenza

Quando una connessione viene aperta, il server salva una struttura dati come allegato alla SocketChannel creata. Questa struttura dati, chiamata Attachment, non è altro che una coppia <username, buffer>. Quando un utente effettua il login con la socket instaurata in precedenza, il server salva lo username dell'utente. Questo meccanismo permette al server di controllare che il client sia autorizzato a effettuare determinate operazioni sul sistema. Nel momento in cui il client effettua l'operazione di *logout()*, questa variabile viene aggiornata con il valore di stringa vuota, in modo tale da permettere il cambio utente, senza dover necessariamente chiude e riaprire l'applicativo o eliminare e re-instanziare una nuova SocketChannel.

Nel caso in cui avvenisse un crash o una chiusura brusca dell'applicativo client, il server riceve una SocketException, il quale viene catturato e gestito facendo effettuare il logout dell'utente dal

sistema, notifica agli altri utenti che è tornato OFFLINE e di conseguenza viene eliminato la SocketChannel a esso legato. Così facendo si evita che risulti ONLINE quando, in realtà non lo è più, permettendo di all'utente di effettuare nuovamente il login.

2.5 Client

Per il client è stato sviluppato una semplice interfaccia a linea di comando. Si compone di tre classi principali:

- La classe ClientMain funge da interfaccia con l'utente, dove vengono inseriti i comandi e restituiti messaggi di feedback sullo stato dell'operazione.
- La classe ClientService, il quale rappresenta la logica applicativa del client e implementa i metodi per gestire la connessione TCP colserver
- la classe ProjectChatTask che si occupa di ricevere e gestire i messaggi relativi alle chat di progetto

2.6 Persistenza dei dati

La persistenza dei dati viene gestita dalla classe "Database" e dal omonima cartella salvata in memoria permanente, i quali simulano una basi di dati per la creazione, salvataggio, modifica e cancellazione[1] dei dati.

```
database/
│
├── projects/
│   ├── progetto1/
│       ├── card1.json
│       ├── card2.json
│       ├── card3.json
│       ├── info.json
│   ├── progetto2/
│       ├── card1.json
│       ├── card2.json
│       ├── info.json
├── users/
│   ├── usr1.json
│   ├── usr2.json
│   ├── usr3.json
```

Figura 3: Esempio di struttura della directory database

Al momento dell'inizializzazione, il server recupera tutti i dati dalla cartella database e li carica in sistema con le seguenti operazioni della classe Database:

- **readFile(file, buffer):** il quale non fa altro che aprire il file e scrivere il contenuto in un ByteBuffer
- **loadProject()**, **loadCard()**, **loadUSer():** i quali prendono il buffer ricavato dall'operazione readFile() e fa la de-serializzazione delle varie strutture dati.

Mentre per ogni richiesta di creazione, modifica e cancellazione da parte del client, viene invocato le seguenti operazioni:

[1] cancellazione ove consentito

- **storeFile(file, fileByte):** il quale apre il file e ci salva il fileByte racchiuso in un ByteBuffer
- le controparti **storeProject()**, **storeCard()**, **storeUser:()** servono per serializzare e mandare in esecuzione lo storeFile()

Ogni progetto ha una cartella dedicata. Al suo interno troviamo:

- **File "info.json":** dove sono salvate le informazioni generali del progetto (nome, lista dei membri, liste delle cards e data di creazione)
- **Una serie di file denominati card*.json:** dove sono salvate le informazioni specifiche della carta (nome, descrizione, stato attuale, lista dei movimenti)

NOTA: Al momento del salvataggio del progetto in memoria permanente vengono ignorati i campi indirizzo IP della chat Multicast. Questo campo, infatti, viene assegnato dinamicamente in fase di inizializzazione del server (se si tratta di progetti esistenti) oppure al momento della creazione dell'istanza di un nuovo progetto.

Per ogni utente è associato un file "*username*.json" dove sono salvate le informazioni che lo caratterizzano (nome, hash della password, salt utilizzato).

Dopo la fase di inizializzazione, il server sarà pronto a eseguire le operazioni richiesti dai client. Ogni qualvolta che il server riceve un comando di creazione o modifica di un dato[2], questa modifica viene apportato in modo sincrono anche nella nella memoria permanente, in modo tale da avere il "database" sempre aggiornato e senza inconsistenze.

2.7 Cifratura delle password

Per garantire un livello di sicurezza aggiuntivo, la registrazione di un utente al servizio include la cifratura della sua password mediante l'algoritmo SHA3-256 e seed casuale: anche se due utenti dovessero utilizzare la stessa password per accedere al servizio, i message digest prodotti saranno differenti tra loro. Ovviamente, per permettere che l'utente possa effettuare il login, il sistema memorizza, per ognuno di loro, la tripla <nome utente, hash password, salt utilizzato>.

2.8 Serializzazione

Tutte le risorse che viaggiano sulla connessione TCP tra client e server e tutti i file salvati sulla memoria permanente del server sono dati struttura secondo il formato standard JSON. Le procedure di serializzazione e de-serializzazione sono effettuati mediante l'ausilio della libreria esterna Jackson versione 2.9.7 (come da lezione).

[2]del tipo *createProject()*, *addMember()*, *moveCard()*, *etc*

3 Schema dei threads avviati

3.1 Lato Server

- **ServerMain:** classe main che instanzia gli altri thread e si preoccupa della fase di inizializzazione dello stato del server
- **RMITask:** thread che si preoccupa di pubblicare gli stub di registrazione RMI e Callback RMI sul Registry
- **SelectionTask:** server che adempie i compiti di accettare nuove connessioni TCP e comunicare con i client

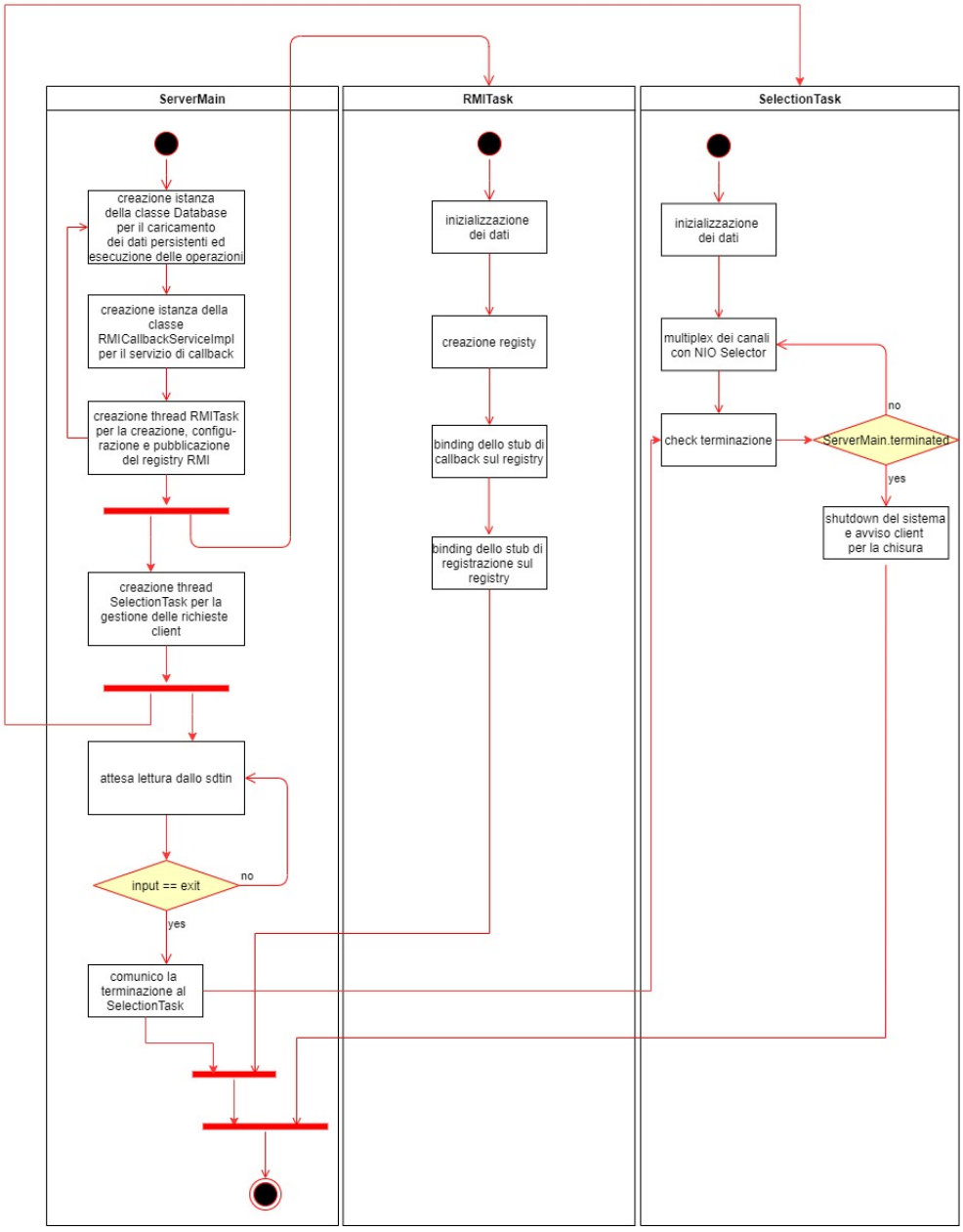

Figura 4: Schema di attivazione thread e descrizione dei loro compiti (server)

Come già anticipato, l'implementazione del server è stata realizzata mediante NIO Selector e SocketChannels. Essendo un server sequenziale, la questione della concorrenza e dell'integrità dei

dati diviene una problematica molto meno pesante. Infatti, tutto ciò che riguarda i progetti non è in alcun modo messo a rischio da eventuali operazioni concorrenti.

Gli unici dati condivisi tra più entità sono quelli riguardanti gli utenti. Questo perché la fase di registrazione è affidata al meccanismo RMI mentre la fase di login è dovuta alla comunicazione TCP esplicita. Per tale ragione, la struttura degli utenti è implementata con una collezione ConcurrentHashMap<String, User>.

3.2 Lato Client

- **ClientMain:** classe main che instanzia la logica applicativa ClientService e gestisce l'interazione con l'utente
- **ProjectChatTask:** thread che si occupa di mettersi in ricezione dei datagrammi UDP (chat). Viene attivato al momento del login e rimane attivo per tutta la durata della sessione dell'utente.

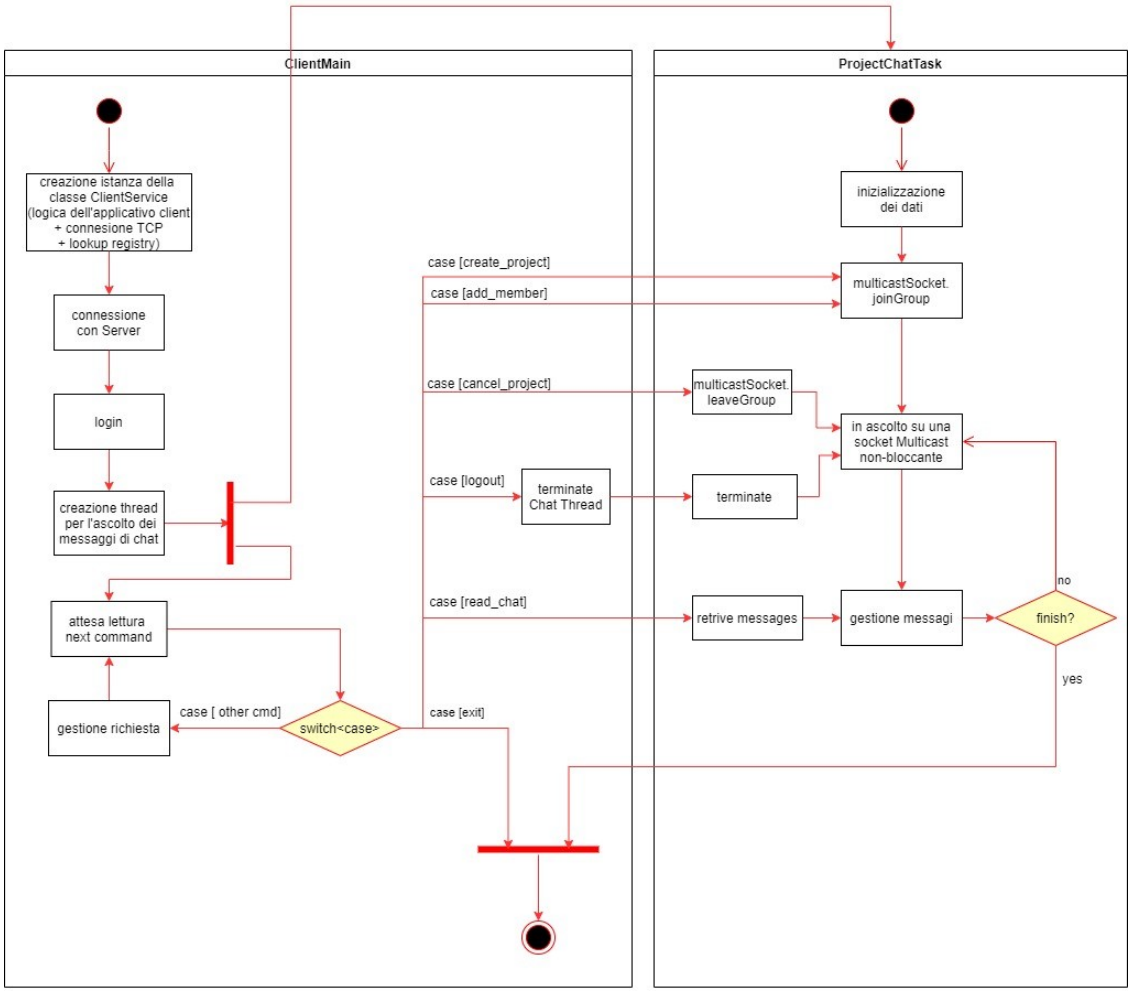

Figura 5: Schema di attivazione thread e descrizione dei loro compiti (client)

4 Classi e interfacce definite

```
worth/
|
├── client/
|   ├── ClientService
|   ├── ProjectChatTask
|   ├── RMICallbackNotify
|   ├── RMICallbackNotifyImpl
|
├── exceptions/
|   ├── tutte le eccezioni definite per il progetto
|
├── data/
|   ├── Card
|   ├── CardImpl
|   ├── CardStatus
|   ├── Movement
|   ├── Project
|   ├── USer
|   ├── UserStatus
|
├── server/
|   ├── RMIOperations/
|       ├── RMICallbackService
|       ├── RMICallbackServiceImpl
|       ├── RMIRegistrationService
|       ├── RMIRegistrationServiceImpl
|       ├── RMITask
|   ├── TCPOperations/
|       ├── Attachment
|       ├── Databse
|       ├── SelectionTask
|       ├── TCPOperation
|       ├── UserRegistration
|
├── utils/
|   ├── CommunicationProtocol
|   ├── ErrorMSG
|   ├── MulticastAddressManager
|   ├── MyObjectMapper
|   ├── PasswordManager
|   ├── PortManager
|   ├── RequestMessage
|   ├── ResponseMessage
|   ├── SuccessMSG
|   ├── UDPMessage
|
├── ClientMain
├── ServerMain
```

Figura 6: La struttura delle classi e interfacce definite nel progetto

4.1 ClientService

Logica dell'applicativo a livello client. Si occupa di comunicare con il server inviando le richieste e ricevendo le risposte sulla connessione instaurata. Contiene le seguenti strutture dati:

- **isLogged:** un flag booleano che indica se sono loggato al sistema
- **username:** il nome utente con cui sono loggato (Stringa vuota in caso non lo fossi)
- **socket:** una socketChannel per la connessione TCP con il server
- **mapper:** Jackson ObjectMapper per la serializzazione e de-serializzazione
- **userStatus:** una HashMap<String, UserStatus> che traccia dei utenti registrati al servizio e i loro relativi stati, il quale viene aggiornata dal server con RMI Callback
- **callbackNotify:** istanza della classe RMICallbackNotify per la registrazione e cancellazione dal servizio RMI callback dei cambiamenti di stato dei utenti e notifica di nuovo progetto
- **projectChat:** istanza della classe ProjectChatTask per la gestione e l'ascolto dei messaggi UDP delle chat dei progetti attivi

La classe oltre ad implementare i metodi richiesti dalle specifiche (del lato client), implementa, per uso interno, una serie di metodi di supporto che sono:

- **public void closeConnection():** metodo chiamato dal *ClientMain* a seguito del comando *exit* inserito dall'utente per terminare l'applicativo. Se l'utente non avesse preventivamente fatto *logout*, il sistema lo fa al posto suo. Il comando exit viene propagato fino al server e dopodiché da entrambe le parti viene chiuso la **socket** per la **connessione TCP**. Dopodiché, fa l'**unexport** degli **oggetti RMI**.
- **private boolean isValid(CardStatus test):** metodo chiamato dall'operazione *MoveCard(projectName, CardName, from, to)* per verificare che i valori *from* e *to* inseriti dall'utente sono valori validi
- **private ResponseMessage sendTCPRequest(requestMessage):** metodo usato da tutte le operazioni che hanno bisogno di comunicare con il server sulla **connessione TCP**. I messaggi vengono prima serializzati in Stringa **JSON**, poi convertiti in array di byte, impacchettati in un buffer e mandati sulla socket di comunicazione, dopodiché aspetta la risposta dal server che può essere positivo, in questo caso si ottiene quanto richiesto, o in caso contrario, si ottiene un messaggio errore.
- **private void registrerForCallback():** metodo chiamato al momento del *login()* per registrarsi al servizio di callback di aggiornamento stato dei utenti, notifica di un nuovo progetto ed arresto del server
- **private void unregisterForCallBack():** metodo chiamato al momento del *logout()* o *closeConnection()* per cancellarsi dal servizio di callback

4.2 ProjectChatTask

Classe che implementa l'interfaccia Runnable, serve per gestire la ricezione dei messaggi (in background) delle chat. È composto dalle seguenti strutture dati:

- **multicastSocket:** istanza della classe MulticastSocket da cui riceve i messaggi UDP
- **chatAddresses:** una mappa dove l'insieme delle chiavi sono i nomi dei progetti che l'utente è membro (aka chat attivi), mentre chatAddresses(k) è l'indirizzo Multicast associato al progetto k.

- **messagges:** una mappa dove l'insieme delle chiavi sono i nomi dei progetti, mentre messages(k) è la LinkedList<UDPMessage> a cui viene concatenato ogni nuovo messaggio ricevuto per il progetto k.
- **finish:** flag booleano per il ciclo infinito della task (all'inizio viene settato a *false*)

Siccome la classe implementa l'interfaccia Runnable fornisce, pertanto, un'implementazione del metodo run(), il quale non fa altro che un ciclo infinito sulle operazioni di ricezione e gestione messaggio. In più implementa le seguenti operazioni:

- i vari metodi get come **getMulticastSocket()**, **getChatAddress(String projectName)**, **getMessages(String projectName)**
- **public void joinGroup(String projectName, InetAddress address):** per unire la multicastSocket ad un nuovo gruppo
- **public void leaveGroup(String projectName):** per abbandonare la multicastSocket da un gruppo precedentemente unito
- **public void terminate():** operazione che permette di bloccare il loop infinito di ascolto dei messaggi dalla multicastSocket e procedere alla chiamata di *leaveGroup()* su tutti i gruppi precedentemente uniti
- **public boolean isTerminated(String projectName):** operazione che verifica se una chat, riconosciuta attraverso il projectName, è ancora attivo o meno. Se una chat è già terminato, permette comunque all'utente di leggere gli ultimi messaggi già ricevuti, ma non ancora letti, prima di distruggerlo completamente.

4.3 RMICallbackNotify

Interfaccia lato client del servizio RMI Callback.

4.4 RMICallbackNotifyImpl

Implementazione dell'interfaccia RMICallbackNotify. È composto dalle seguenti strutture dati:

- **userStatus:** mappa concorrente dove l'insieme delle chiavi sono gli username degli utenti registrati, mentre userStatus(k) è lo stato del utente k
- **projectChats:** mappa dove l'insieme delle chiavi sono tutti gli projectName che l'utente è membro, mentre projectChat(k) è la chat multicast associato al progetto k

Le operazioni disponibili sono:

- **public synchronized void notifyUSerStatus(username, status):** operazione che notifica al client di un nuovo cambiamento di stato dell'utente *username*
- **public synchronized void notifyNewProject(projectName, chatAddress):** operazione che notifica che l'utente è stato aggiunto ad un nuovo progetto e di conseguenza viene fatto la joinGroup sul nuovo indirizzo Multicast comunicato.
- **public void notifyCloseClient():** operazione che notifica l'utente dell'arresto del server e invita l'utente di eseguire l'arresto manuale anche del client.
- **public void leaveMulticastGroup(String projectName):** operazione che viene lanciato a seguito della cancellazione di un progetto e quindi abbandona il gruppo Multicast del progetto interessato.

4.5 Card

Interfaccia che definisce il comportamento di una Card che non possiede la lista dei movimenti. Utilizzata per serializzare una Card senza i movimenti quando l'utente richiede il servizio showCard.

4.6 CardImpl

Rappresenta una card di un progetto. Implementa le interfacce **Serializable** e **Card**. I suoi attributi sono **name**, **description**, **status** e **list of moviments**

E implementa i seguenti metodi:

- i vari metodi get come **getName()**, **getDescription()**, **getStatus()**, **getMovements()**
- **public void changeStatus(*newStatus*):** modifica lo stato del card in *newStatus*
- @Override del metodo **equals()**

4.7 CardStatus

Rappresenta l'enumerazione dei possibili stati di un card: enum {TODO, INPROGRESS, TOBEREVISED, DONE}. Implementa l'interfaccia **Serializable** in modo da poter rendere le sue istanze persistenti in memoria.

E implementa il seguente metodo:

- **public static CardStatus retriveFromString(String stringStatus):** restituisce il valore corrispondente alla stringa passata come argomento.

4.8 Movement

Rappresenta il movimento di una card. È una tripla <*stato di partenza*, *stato di arrivo*, *data dell'avvenimento*>. Implementa l'interfaccia **Serializable** in modo da poter rendere le sue istanze persistenti in memoria.

4.9 Project

Rappresenta un progetto all'interno del servizio WORTH ed implementa l'interfaccia **Serializable**. Possiede un nome, una lista di membri (solo *username*), data di creazione, liste di stati delle card (per ogni stato sono memorizzati i nomi delle card che ne fanno parte), indirizzo IP Multicast della chat, lista di tutte le card (oggetti Card). Gli ultimi 2 attributi vengono ignorati al momento della serializzazione poiché ogni card del progetto viene serializzato in un file a parte, mentre l'indirizzo non deve essere persistente, ma assegnato dinamicamente al momento del riavvio del server.

Comprende i seguenti metodi:

- **public void initChatAddress(String address):** metodo utilizzato dal server per inizializzare l'indirizzo multicast del progetto al momento della de-serializzazione
- **public void initChatPort(int port):** metodo utilizzato dal server per inizializzare la porta multicast del progetto al momento della de-serializzazione
- **public void initCardList(List<CardImpl> cards):** metodo chiamato dal server per inizializzare la lista delle cards precedentemente de-serializzate
- **public void moveCard(String cardName, CardStatus from, CardStatus to):** metodo utilizzato per spostare la card *cardName* dallo stato *from* allo stato *to*.
- **public void addMember(String user):** operazione per l'aggiunta di un nuovo membro al progetto

- **public boolean isCancelable():** l'operazione restituisce true sse tutte le card del progetto sono nello stato DONE
- **private boolean moveIsAllowed(CardStatus from, CardStatus to):** chiamato da *moveCard()* verifica se l'operazione di spostare la card dallo stato *from* allo stato *to* rispetta i vincoli delle specifiche date.
- **private boolean isValid(CardStatus test):** verifica lo stato *test* dell'argomento è uno stato valido $\in$ enum {TODO, INPROGRESS, TOBEREVISED, DONE}

4.10 User

Rappresenta l'utente del servizio caratterizzato dal username, hash della password e il salt. la classe implementa l'interfaccia **Serializable** in modo da poter rendere le sue istanze persistenti in memoria.

4.11 UserStatus

Rappresenta l'enumerazione dei possibili stati di un utente registrato: enum {ONLINE, OFFLINE}. Implementa l'interfaccia **Serializable** in modo da poter rendere le sue istanze persistenti in memoria.

4.12 RMICallbackService

Interfaccia lato server del servizio RMI Callback.

4.13 RMICallbackServiceImpl

Implementazione dell'interfaccia lato server del servizio RMI Callback. È composto dalle seguenti strutture dati:

- **clients:** ConcurrentHashMap dove l'insieme delle chiavi sono gli username degli utenti registrati al RMI Callback, mentre clients(k) è il client associato all'utente k. Si è preferito questa struttura dati poiché è **tread-safe** e non è ammesso chiavi o valori nulli
- **toDelete:** ArrayList di username dei client da de-registrare. Questo è pensato quando un client viene interrotto bruscamente e di conseguenza, non farà mai la unregister dal servizio rmi. Al momento dell'invio di una notifica, il server riceve un errore di connessione con il suddetto client, per tale ragione, ogni volta che si riceve un errore di questo genere, il client verrà inserito in questa lista e de-registrato dal servizio di callback

Comprende i seguenti metodi:

- **registrazione** e **de-registrazione** dal servizio
- **public synchronized void notifyUsers(String username, UserStatus status):** notifica ogni client registrato del cambio stato dell'utente username, nel caso di errori di connessione con alcuni client, questi vengono aggiunti alla list toDelete e poi rimossi.
- **public synchronized void notifyProject(String username, String projectName, String chatAddress):** notifica all'utente interessato che è stato aggiunto come membro ad un nuovo progetto e gli viene fornito anche indirizzo della chat multicast
- **public void notifyServerDown():** notifica a tutti i client che il server si sta arrestando
- **public void terminateChat(String projectName, List<String> members):** notifica ai membri del progetto projectName che devono abbandonare il gruppo Multicast associato a quel progetto

4.14 RMIRegistrationService

Interfaccia del servizio di registrazione al servizio WORTH via metodo RMI.

4.15 RMIRegistrationImpl

Implementazione dell'interfaccia RMIRegistrationService. Permette la registrazione di un nuovo utente al servizio e la notifica dello stesso agli altri utenti via RMI Callback

4.16 RMITask

Task per la creazione e la pubblicazione del Regitry RMI dei servizi di registrazione e callback.

4.17 Attachment

Rappresentazione dell'attachment di una socket nel selector, ha gli attributi username e buffer.

4.18 Database

Questa classe si occupa di realizzare le funzionalità di WORTH. È lei che possiede tutte le strutture dati su cui si basa il servizio, che implementa le operazioni richieste dalle specifiche e gestisce la salvataggio e caricamento dei dati in/da memoria persistente. Implementa le interfacce UserRegistration e TCPOperations. Fa uso delle seguenti strutture dati:

- **Users:** mappa concorrente dove l'insieme delle chiavi sono gli username degli utenti registrati, mentre users(k) è l'istanza del singolo utente k. Unica struttura dati ad essere acceduta concorrentemente nel sistema
- **Projects:** mappa dove l'insieme delle chiavi sono i nomi dei progetti nel sistema, mentre projects(k) è l'istanza del singolo progetto k.
- **userStatus:** mappa dove l'insieme delle chiavi sono gli username degli utenti registrati, mentre userStatus(k) è lo stato attuale dell'utente k (online, offline)

4.19 SelectionTask

Task che si occupa della gestione delle connessioni, mediante il Multiplexing dei canali, e della comunicazione con i clients. Al momento della creazione gli viene passato un'istanza della clasee Database per la gestione delle operazioni TCP e un'oggetto della classe RMICallbackServiceImpl per la gestione delle operazioni di callback, si instanzia poi un ObjectMapper per la serializzazione/de-serializzazione dei dati da mandare/ricevere sulla connessione TCP instaurata e la gestione del servizio callback.

L'implementazione del NIO Multiplexing è basato su 3 componenti fondamentali:

- un oggetto deela classe ServerSocketChannel che rimane in ascolto delle connessini in arrivo;
- un SocketChannel per ogni coneessione attiva;
- un Selector per iterare sulle connessione attive ed effettuare il multiplexing di queste

Al momento dell'avvio viene istanziato un Selector e il ServerSocketChannel, il quale viene settato in modalità non-bloccante per il corretto funzionamento del selettore. A seguito della registrazione della ServerSocketChannel sull'operazione ACCEPT, il task entra in un ciclo infinito, dove ad ogni iterazione verifica se ci sono canali pronti per effettuare qualche tipo di operazione. Si crea così un oggetto di tipo Iterator<SelectionKey> per iterare sul set di chiavi corrispondenti ai canali disponibili. Per ogni chiave restituita dall'iteratore, si individua il tipo di evento pronto da essere elaborato:

- se la chiave corrisponde ad un evento di tipo OP_ACCEPT, vi è una nuova connessione in arrivo e si invoca quindi l'operazione accept() sulla socket, il quale si occupa di creare un nuovo SocketChannel per la connessione e dopodiché, si registra che il canale è pronto per le operazioni di lettura.
- se la chiave corrisponde ad un evento di tipo OP_READABLE, il programma si fa restituire il canale dalla chiave associata. Se si riscontrasse errori e il canale non fosse leggibile, viene lanciato una SocketException, il quale sta a indicare che il client non è più raggiungibile. In questo caso se l'utente associato risulta ancora online viene fatto il logout e dopodiché viene eseguito la chiusura della socket e l'eliminazione della chiave. Nel caso fosse andato tutto liscio, viene istanziato la struttura dati Attachment (già illustrato in precedenza) e si procede con la lettura dei dati dal buffer: il contenuto del buffer viene decodificato in Stringa, e a seguire, de-serializzato in RequestMessage. Da quest'ultimo viene ricavato il comando dell'operazione richiesta dal client e si procede con il match delle operazioni disponibili dal server. Se l'operazione va a buon fine o incontra un errore, il ResponseMessage viene preparato con lo statusCode opportuno ed eventuale responseBody con la struttura richiesta e si procede con le operazioni inverse (serializzazione e codifica in bytes). Fatto questo si prepara la chiave del canale sull'operazione di scrittura.
- se la chiave corrisponde ad un evento di tipo OP_WRITABLE, si prende il buffer dall'Attachment e si passa all'operazione di scrittura sulla socket.

4.20 TCPOperation

Interfaccia che definisce le operazioni da far transitare sulla connessione TCP

4.21 UserRegistration

Interfaccia che definisce l'operazione di registrazione offerta

4.22 CommunicationProtocol

Fondamenta del progetto. Qui sono descritti tutti i comandi e gli status code che possono essere prodotti.

4.23 ErrorMSG

Classe astratta dove sono elencati tutti i messaggi di errore previsti dal sistema

4.24 MulticastAddressManager

Classe astratta che gestisce gli indirizzi Multicast necessari al funzionamento della chat. Genera indirizzi Multicast in sequenza da 239.0.0.0 a 239.255.255.255 (indirizzi Multicast locali). Utilizzata per attribuire ad un progetto un indirizzo Multicast. Non mantiene alcuna informazione in memoria persistente, gli indirizzi sono riassegnati dall'inizio ogni volta il server viene riavviato. Quando un progetto viene cancellato, l'indirizzo viene liberato per essere riutilizzati in un nuovo progetto.

4.25 MyObjectMapper

Jackson ObjectMapper per la serializzazione e de-serializzazione dei oggetti con funzionalità estese come l'indentazione e il formato della data.

4.26 PasswordManager

Classe che gestisce la cifratura e il confronto delle password, utilizza la funzione di cifratura SHA-256 con salt.

4.27 RequestMessage

Struttura di messaggio di richiesta del client. Contiene i campi:

- **command:** campo contenente il nome dell'operazione richiesta
- **arguments:** gli argomenti necessari all'operazione

4.28 ResponseMessage

Struttura di messaggio di risposta del server. Contiene i campi:

- **statusCode:** codice di stato del risultato dell'operazione
- **responseBody:** i dati richiesti dall'operazione (può essere null)
- **responseBody2:** campo utilizzato solo dall'operazione di *loging()* per l'invio degli indirizzi delle chat di progetto. Per tutte le altre operazioni il campo è null

4.29 SuccessMSG

Classe astratta dove sono elencati tutti i messaggi di successo previsti dal sistema

4.30 UDPMessage

Struttura di un datagramma UDP. Contiene i campi:

- **author:** username di chi ha scritto il messaggio
- **message:** testo del messaggio
- **projectName:** il progetto a cui fa riferimento
- **fromSystem:** campo booleano settato solamente dal server per inviare messaggi di sistema sulla chat

5 Istruzione di compilazione

Istruzioni di compilazione per l'ambiente **Windows**. Assicurarsi di essere dentro la directory principale del progetto (dove sono src/, lib/, ...).

Da linea di comando:

- Crea un file dove sono elencati tutti i file da compilare con javac, con il seguente comando:

  ```
  $ dir /s /B *.java > ./paths.txt
  ```

- Genera bytecode

  ```
  $ javac -d "bin" -cp lib/jackson-annotations-2.9.7.jar;lib/jackson-core-2.9.7
  .jar;lib/jackson-databind-2.9.7.jar;lib/jackson-datatype-jsr310-2.9.7.jar @paths.txt
  ```

- Esegui server

  ```
  $ java -cp lib/jackson-annotations-2.9.7.jar; lib/jackson-core-2.9.7.jar;lib/
  jackson-databind-2.9.7 .jar;lib/jackson-datatype-jsr310-2.9.7.jar;bin com.ServerMain
  ```

- Esegui client

  ```
  $ java -cp lib/jackson-annotations-2.9.7.jar;lib/jackson-core-2.9.7.jar; lib/
  jackson-databind-2.9.7.jar;lib/jackson-datatype-jsr310- 2.9.7.jar;bin com.ClientMain
  ```

NB: Attenzione ad eventuale spazio non dovuto quando si fa copia/incolla dei comandi dal pdf a linea di comando