UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

LABORATORIO DI RETI (2018-2019)

TURING: disTribUted collaboRative edItiNG

Autore:

Giacomo Peparaio

Indice

Part I
Architettura del sistema

1 Organizzazione del codice

Il codice è strutturato in 3 package principali: Client, Server e Common.
I primi due sono a loro volta divisi secondo il pattern architetturale "Model-View-Controller", Common invece contiene il codice comune al Client e al Server.

1.1 Package client

1.1.1 Package client.view

Per facilitare l'utilizzo ho deciso di implementare il client con l'interfaccia grafica(Swing). Questo package contiene il codice dei 3 JFrame utilizzati:

- *Login*: registrazione utente e login.
- *Logged*: funzionalità offerte ad un utente online(newDoc,share,show,edit,notifiche inviti).
- *Editing*: servizio di chat durante la fase di editing.

1.1.2 Package client.controller

Composto da classi che controllano la view e comunicano con il server:

- *TuringClient*: contiene Main del client.
- *ControlListener*: classe "listener" degli eventi provenienti dalla view, si occupa di gestirli e di aggiornare l'interfaccia grafica in modo opportuno.
- *ListingThread*: Thread che aggiorna la lista dei files modificabili da un utente sulla view.
- *ChatThread*: Thread in attesa di nuovi messaggi dalla chat, si occupa anche lui di aggiornare la view.

1.2 Package server

1.2.1 Package server.model

Contiene classi che modellano la memoria del server, ovvero l'organizzazione e lo stato di documenti e utenti:

- *Utente*: modella l'utente del servizio.
- *Documento*: modella i Documenti.
- *DataBase*: struttura dati principale del server, utilizza le classi precedenti per modellare lo stato. Poiché deve essere unica è stata strutturata secondo il design pattern Singleton.

1.2.2 Package server.controller

Composto da classi che interagiscono con i client e gestiscono il DataBase.

- *RegistrationImpl*: Implementa la RegistrationInterface, ed appunto fa parte del servizio RMI per la registrazione di un nuovo utente.
- *TuringServer*: contiene il Main del server, inizializza il servizio Rmi e fa da thread listener per nuove connessioni dai client.
- *RequestHandlerThread*: classe più importante del server, implementa il thread che gestisce le richieste del client.

1.3 Package common

Codice comune a client e server, è composto dalle seguenti classi:

- *RegistrationInterface*: registrazione con RMI.
- *Configuration*: configurazione del sistema(indirizzi IP, porte utilizzate, path).
- *Packet*: pacchetto utilizzato per la comunicazione tra client e server.
- *typePack*: enumerazione che codifica il tipo del pacchetto(errore, richiesta...).

2 Strutture dati

2.1 Classe DataBase

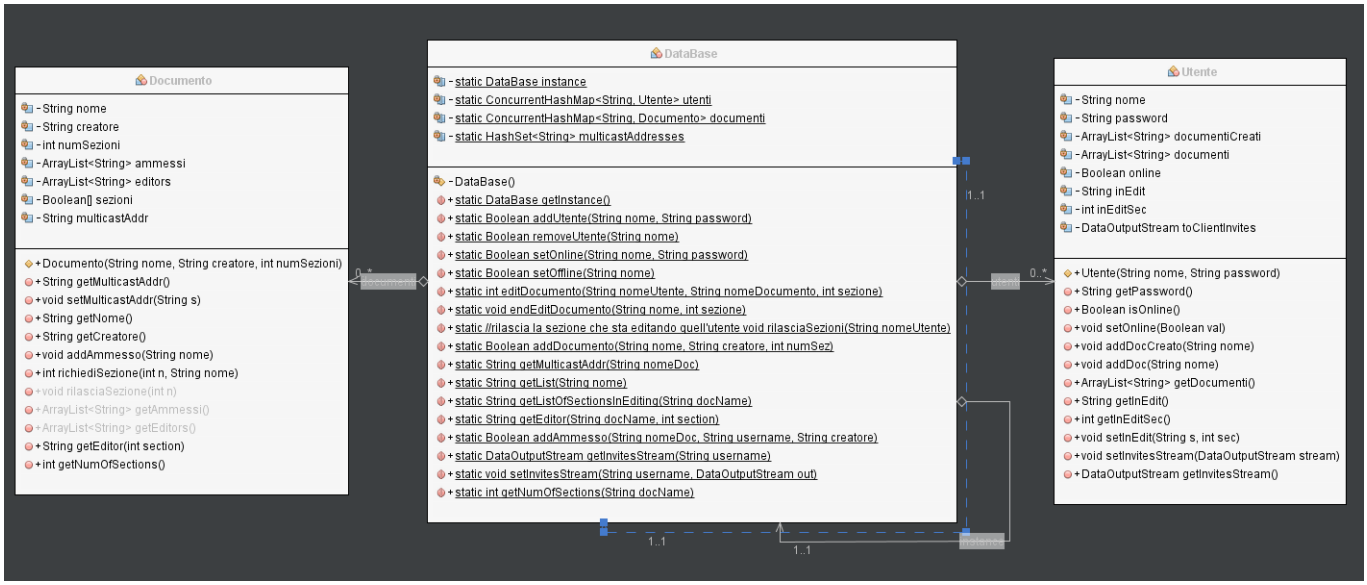
Questa classe utilizza 2 ConcurrentHashMap:

- *utenti*: Coppia <String, Utente>dove come chiave ho il nome utente che deve essere unico, e come valore ho un oggetto Utente. Contiene tutti gli utenti iscritti al servizio.
- *documenti*: Coppia <String, Documento>qui come chiave ho il nome del documento(anch'esso unico) e come valore un oggetto Documento. Contiene tutti i documenti creati.

ho scelto la ConcurrentHashmap poiché avevo bisogno di strutture ThreadSafe(il server è multithreaded), ed efficienti in tempo(non avrei le stesse prestazioni con una lock su tutta la struttura) poiché sono molto sollecitate a Runtime.

Inoltre è presente una terza struttura(HashSet), meno importante delle precedenti, per la memorizzazione degli indirizzi Multicast. Quando viene modificata qui la lock avviene sull'oggetto, ma non causa ritardi poiché la struttura è poco sollecitata a Runtime.

2.2 Classi Utente e Documento



Utente e Documento sono utilizzate solo da DataBase e risultano trasparenti a chi utilizza la struttura, questo garantisce una maggiore robustezza del codice.

Nella classe Utente le strutture degne di nota sono documentiCreati(lista nomi documenti creati dall'utente), e documenti(quelli a cui è stato invitato). InEdit ed inEditSec contengono rispettivamente nome Documento che l'utente sta editando e relativa sezione(NB: le sezioni partono da 0). Ultima variabile importante è toClientInvites(stream per notificare gli inviti al client).

Invece in Documento ho ammessi(lista nomi degli utenti ammessi a modificarlo), editors(lista di chi lo sta editando in quel momento), sezioni(ogni cella indica se la sezione i-esima è in-editing), e multicastAddr(indirizzo multicast relativo alla chat del documento).

2.3 Concorrenza

Utente e Documento non sono Thread-Safe, quindi nei metodi di DataBase, quando accedo alle loro istanze, acquisisco la lock su tutto l'oggetto(synchronized(obj)).

Più thread simultaneamente possono mandare gli inviti allo stesso client, quindi le scritture sullo stream toClientInvites sono atomiche.

Part II
Runtime

In seguito è descritto il ciclo di esecuzione di client e server e i vari thread attivati.

3 Lato Server

3.1 Thread Main (TuringClient.java)

Qui avviene l'esportazione dell'oggetto RegistrationImpl, poi si entra in un ciclo while(true) dove ad ogni iterazione viene accettata una connessione TCP su un ServerSocket, la quale poi verrà passata al costruttore di RequestHandlerThread. Quindi per ogni connessione vado a crearmi un task che gestirà le richieste del client fino al logout(connessioni persistenti).

I task sono eseguiti da un FixedThreadpool : è indicato nel caso di CPU intensive tasks, quindi avendo connessioni persistenti fa al caso nostro. Ho deciso di fissare il numero dei Thread in modo da renderlo modificabile dal file di configurazione a seconda delle risorse che abbiamo a disposizione.

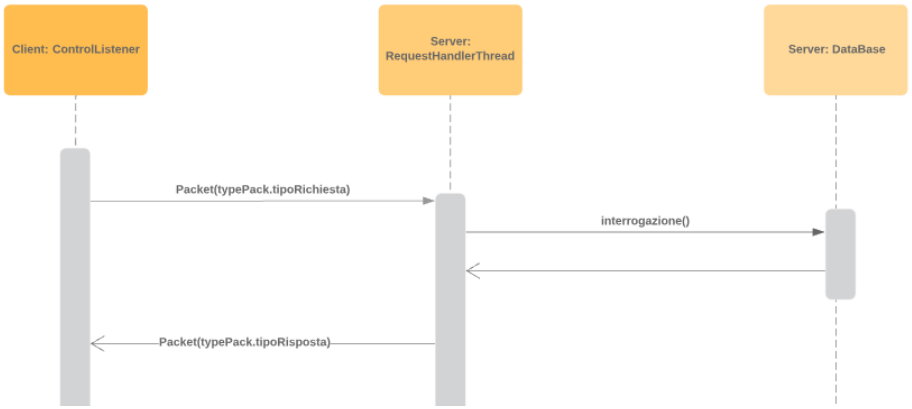
3.2 Thread Richieste(RequestHandlerThread.java)

Come detto sopra ogni sua istanza corrisponde ad un singolo client, e gestirà tutte le richieste inviate da esso. Sostanzialmente si tratta di un while(true) dove ad ogni iterazione aspetta di leggere un Packet dalla connessione TCP passata al costruttore. A seconda del tipo di pacchetto gestisce la richiesta relativa (interrogando DataBase se necessario) e spedisce il packet risposta sulla stessa connessione.
Se viene rilevata una eccezione o l'utente effettua il logout, prima di terminare il task, i documenti lockati dal client vengono rilasciati e l'utente viene settato offline.
Quando un utente crea un documento, o viene invitato, questo thread si occupa anche di inviare le notifiche sullo stream del client corrispondente.

4 Comunicazione

Dopo il login, tra client e server, saranno attive rispettivamente 2 connessioni TCP persistenti e una non persistente:

- *Socket richieste*: Qui vengono inviate le richieste e le risposte tra client e server sotto forma di Packet, la comunicazione è sincrona.(vedi immagine)
- *Socket inviti*: Qui passano le notifiche degli inviti quando il client è online.
- *SocketChannel documenti*: Viene creata ogni volta che client e server devono scambiarsi un file(Java NIO).

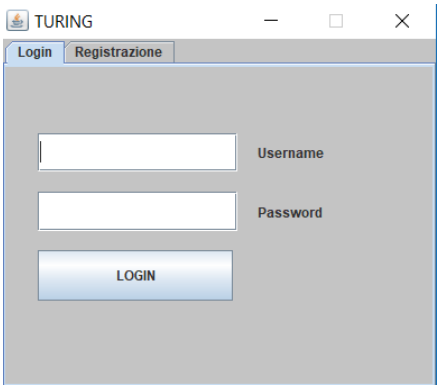


I pacchetti sono in formato JSON(libreria json Simple), ho fatto questa scelta per essere il più standard possibile. Come ultima cosa vorrei aggiungere che il servizio di chat è Peer2Peer, quindi i messaggi non passano per il server.

5 Lato Client

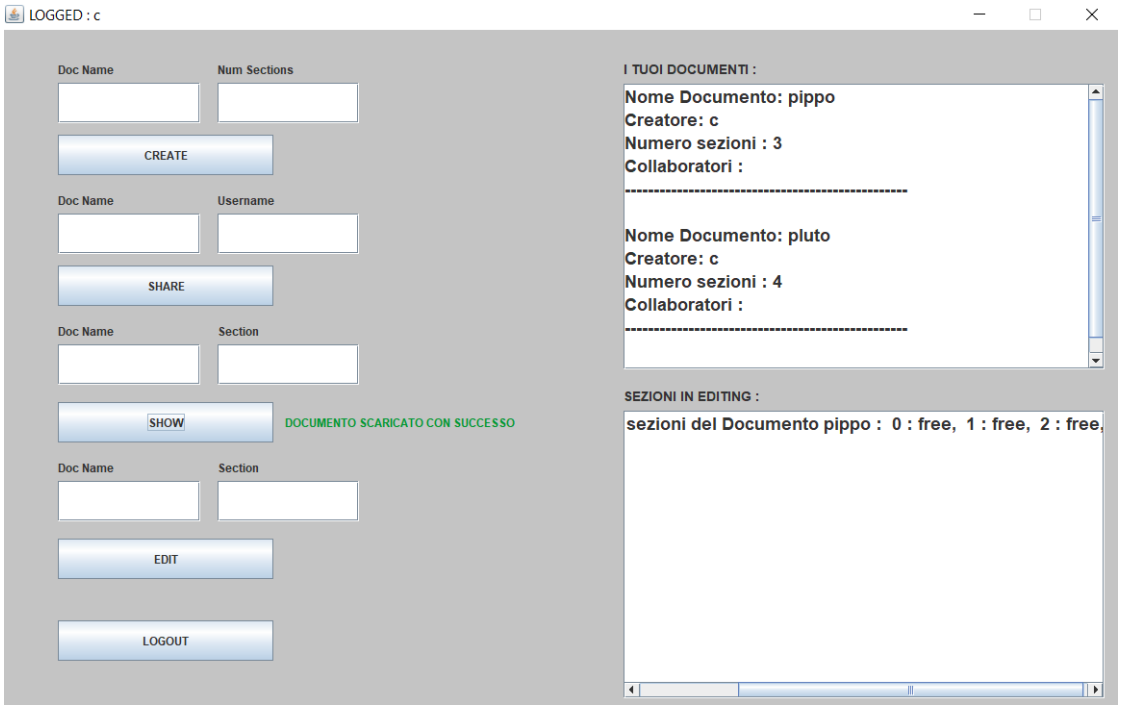
Breve descrizione dei tre JFrame del client, è bene ricordarsi che tutti i controlli della vista hanno registrato come ActionListener la classe "ControlListener".

5.1 Registrazione e Login (Login.java)



Schermata di login, niente di rilevante da aggiungere.

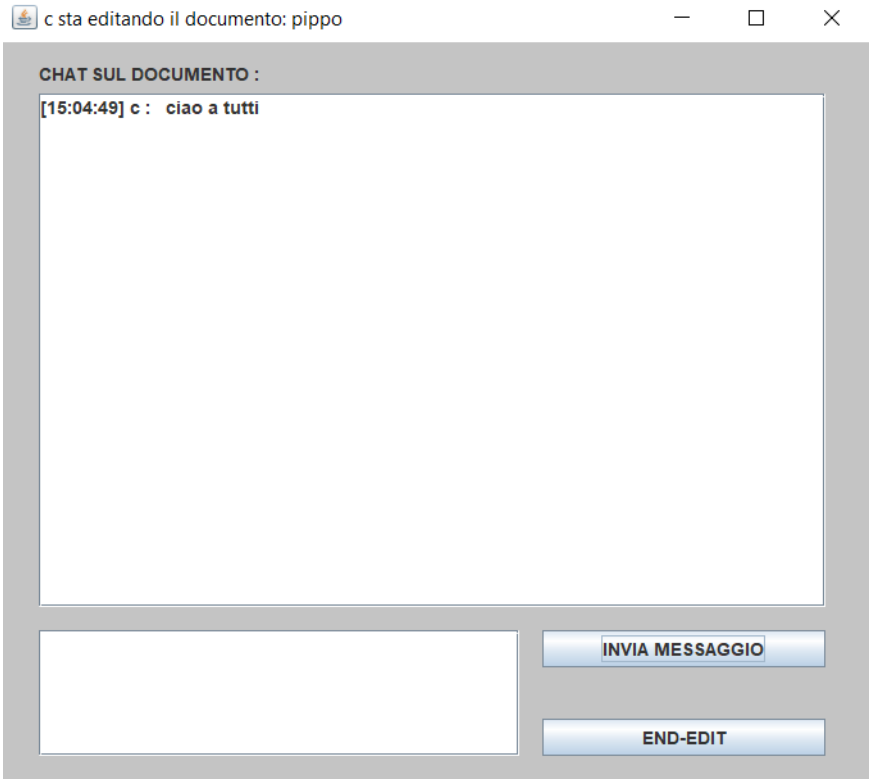
5.2 Utente loggato (Logged.java)



Una volta loggati apparirà questo JFrame. La JTextArea in alto a destra viene aggiornata dall'istanza di ListingThread.java(avviato insieme al JFrame) ogni volta che si crea un documento, o si viene invitati a modificarne uno. Quella in basso a sinistra mostra i risultati del comando Show(quali sezioni sono in-editing e chi sta editando).

Ho cercato di gestire tutte le possibili "cavolate" che un utente potrebbe fare, ma comunque,anche se mi fosse sfuggito qualcosa, il server solleverebbe un' eccezione disconnettendo successivamente il client(quindi turing rimane operativo).

5.3 Utente in-editing (Editing.java)



Durante l'editing di un documento avremo questa schermata. La JTextArea in alto è aggiornata dall'istanza di ChatThread.java(avviato insieme al JFrame) ogni volta che si riceve un messaggio sull'indirizzo multicast del documento. Quella in basso contiene il corpo del messaggio da inviare.

Part III

Testare il Progetto

```
WINDOWS:
javac -classpath ./json-simple-1.1.1.jar ./common/*.java ./client/view/*.java  ./client/controller/*.java ./server/model/*.java ./server/controller/*.java

java -cp ".;./json-simple-1.1.1.jar" server.controller.TuringServer

java -cp ".;./json-simple-1.1.1.jar" client.controller.TuringClient

LINUX:
javac -classpath ./json-simple-1.1.1.jar ./common/*.java ./client/view/*.java  ./client/controller/*.java ./server/model/*.java ./server/controller/*.java

java -cp ".:./json-simple-1.1.1.jar" server.controller.TuringServer

java -cp ".:./json-simple-1.1.1.jar" client.controller.TuringClient
```

Per testare il progetto bisogna aprire un terminale sulla cartella "src" ed eseguire i comandi sopra indicati(si trovano anche nella cartella del progetto). Prima di eseguire il client accertarsi che sia comparsa la scritta "TURING SERVER READY".
I documenti verranno salvati in "turing-docs" per il server e in "client-turing-docs" per il client.
Ricordarsi che il numero di client gestibili è limitato, tale valore è modificabile dal file di configurazione. Inoltre sono modificabili anche i path dove vengono salvati i files e le porte usate per le connessioni.