UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

LABORATORIO DI RETI (2018-2019)

TURING: disTribUted collaboRative edItiNG

Autore:

Giulio Purgatorio

Indice

Part I
Introduzione

Questo documento contiene la relazione per il progetto finale di Reti dell'anno 2018-2019.
Riassumendo enormemente per dare una visione d'insieme di cosa parleremo durante il documento, Turing è uno strumento per l'editing collaborativo di documenti. L'interfaccia grafica è un optional che a mio parere rende più fruibile non solo l'interfacciarsi con il software, ma anche la stesura del codice stesso. Il linguaggio usato è Java, senza librerie esterne particolari.
Premessa doverosa, in quanto non è stato esplicitamente richiesto, ho evitato di gestire la persistenza dei dati: ogni volta che il server viene terminato, ogni azione eseguita non avrà ripercussioni nell'esecuzione successiva. Inoltre, avendo chiesto al docente il permesso a riguardo, non ho gestito "nella maniera corretta" l'interfaccia grafica creando nuovi Threads per ogni richiesta: le richieste vengono quindi fatte dal Thread che gestisce l'interfaccia stessa.

1 Strutture dati utilizzate

Prima di parlare del progetto vero e proprio, ritengo necessario un piccolo excursus per vedere quali strutture dati siano state utilizzate per realizzare il progetto, seguite dalle motivazioni della scelta.

1.1 ConcurrentHashMap

Il server utilizza due ConcurrentHashMap. Ho scelto di usare l'HashMap per via degli apprendimenti acquisiti durante il percorso di laurea. In particolare, essendo Concurrent, comporta che tutte le operazioni siano Thread-safe, in quanto ci saranno ovviamente più utenti che lavorano in cooperativa nello stesso momento. Da notare è che una ConcurrentHashMap non assicura con nessun meccanismo interno (per esempio, lock) che una Write ed una Read siano coerenti, in quanto la Read (retrieval operation) riporta i risultati dell'ultima operazione completata. Le due HashMap hanno lo scopo di gestire le due classi principali di Turing, ovvero la classe Utente (User.java) e la classe Documento (Document.java):

- *database*: Coppia <String, User>che permette la gestione degli Utenti sapendo il loro username.
- *docs*: Coppia <String, Document>che permette la gestione dei Documenti conoscendone il nome.

Essendo le due ConcurrentHashMap "database" e "docs" non oggettivamente collegate tra di loro, può esistere un problema di consistenza nei dati. Infatti, nel caso di necessità di un aggiornamento che preveda lavoro su entrambe le strutture, ciò viene evitato chiamando una synchronized su un oggetto, updateDB, che funge da lock per quando ci sia bisogno di aggiornarle entrambe. Ho scelto di usare una synchro su un Object invece di una lock in modo da mostrare un ulteriore argomento che è stato affrontato durante il corso; le locks invece sono state usate su un punto spiegato più avanti.

1.2 HashSet

Il server utilizza anche due HashSet per gestire gli utenti connessi e non. In breve, quindi, l'unione dei due Sets rappresenta l'insieme degli usernames attualmente iscritti al sistema.
La scelta del Set deriva dal fatto che gli usernames degli utenti sono identificatori univoci e, in quanto tali, non possono essere duplicati. Questa informazione suggerisce quindi l'uso di un Insieme, dove è facile aggiungere, rimuovere e controllare l'esistenza di elementi. Una peculiarità del Set è che non ci sono garanzie su come gli elementi siano ordinati una volta estratti: questo implica che su una grande mole di dati (in questo caso, di utenti registrati) questa scelta debba essere cambiata per questioni d'efficienza, ma per l'esecuzione usuale di questo progetto questa caratteristica non comporta alcun svantaggio. I due Sets sono denominati:

- *usersOnline*: come suggerito dal nome, è l'insieme degli utenti attualmente connessi.
- *usersOffline*: come suggerito dal nome, è l'insieme degli utenti attualmente non connessi.

Un'ultima particolarità è la possibilità di inserire elementi "null" nell'HashSet, seppur sia impossibile farlo nell'esecuzione del progetto normale, in quanto la User Interface che gestisce la richiesta di registrazione, controlla esplicitamente che il campo Username (e Password) non siano null. Per quanto riguarda la coerenza tra i dati presenti nei due Sets, ho preso la decisione di rimanere coerente con la scelta effettuata per le ConcurrentHashMaps, permettendo le operazioni su tali insiemi solamente ottenendo la synchro su un oggetto creato ad hoc, UpdateSets.

2 Classi principali

Essendo il progetto rivolto alla gestione di editing di documenti da parte di utenti registrati, pare evidente la necessità di descrivere le due classi principali.

2.1 Classe Utente: (User.java)

Classe che rappresenta un utente iscritto al servizio Turing. Analizzo in breve scopo ed usi dei vari campi:

- (String) Username: usato sia per accedere al servizio (combinato con il campo Password) sia come identificativo univoco all'interno di Turing. Essendo un identificativo univoco, esso viene usato come argomento in varie operazioni, vedasi Inviti, etc.
- (String) Password: usato per accedere al servizio (combinato con il campo Username). Non ha ulteriori utilizzi se non quelli di una Password convenzionale.
- (Set) userDocs: un HashSet di String che serve come controllo ausiliario per sapere a quali documenti l'utente ha accesso (creatore o editor). Questa nozione verrà ripetuta in Document.
- (Set) pendingInvites: un HashSet di String dove vengono salvati tutti i nomi dei documenti a cui l'utente viene invitato durante il suo periodo di permanenza offline dal servizio. Verranno richiesti ed inviati ogni qualvolta l'utente fa login, con ovvia pulizia del contenuto ad ogni invio.

- (Set) instaInvites: un HashSet di String dove vengono salvati tutti i nomi dei documenti a cui l'utente viene invitato durante il suo periodo di permanenza attiva nel servizio. Verranno richiesti costantemente in modo da ottenere un avvertimento in tempo reale (vedasi la sezione Inviti Live).

2.2 Classe Documento: (Document.java)

Classe che rappresenta un documento registrato sul servizio di Turing. Analizzo in breve scopo ed usi dei vari campi:

- (String) DocName: nome del documento. Funge da identificativo univoco all'interno di Turing, quindi, come l'Username per la classe User, non possono esistere due documenti con lo stesso nome.
- (String) Creator: nome utente di chi ha creato il documento. Serve in quanto solo chi ha creato il documento ha i permessi per invitare altri utenti a modificarlo.
- (List) Editors: LinkedList di String dove possiamo trovare i nomi utente delle persone invitate a modificare il documento.
- (List) Locks: Una lista di ReentrantLocks (più precisamente, un Array.asList() che fa da ponte tra le due implementazioni) a cui ad ogni elemento corrisponde un file.txt. Ciò serve in quanto non più di una persona può modificare una determinata Sezione nello stesso momento. Quando un utente desidera modificare quindi una sezione, può farlo se e solo se tale Lock è libera: in questo modo possiamo notare che questa List simula la rappresentazione stessa delle sezioni.
- (InetAddress) MulticastAddr: indirizzo di Multicast assegnato a tempo di creazione documento da parte del server. Gli utenti useranno questo indirizzo per poter chattare tra di loro durante la fase di modifica di un documento. L'indirizzo è quindi statico, come abbiamo visto durante il corso, in quanto questo rimane assegnato anche se non vi sono partecipanti in un determinato istante. In quanto tale indirizzo deve essere unico, viene salvato anch'esso in un Set a parte in modo da non rischiare di avere più indirizzi uguali in fase di creazione Documento. In particolare, gli indirizzi di Multicast partono da 239.0.0.0: ho scelto questa semplificazione in quanto il numero di indirizzi risultante è più che sufficiente per questo progetto. Ovviamente se il progetto dovesse usare così tanti documenti basterebbe ampliare il range da 224.0.1.0 fino all'attuale limite superiore, in modo da poter utilizzare l'intero range disponibile.

Part II
Ciclo d'Esecuzione

3 Lato Server (Turing.java)

Il server è, come in ogni paradigma Client-Server, il processo che deve essere pronto prima dell'esecuzione di altri clients. Esso ha due fasi, una di inizializzazione ed una di specializzazione. Vorrei specificare che la gestione dei files implicitamente richiede che non vi siano azioni locali da parte di chi gestisce il server che puntino a far fallire le operazioni: la creazione di un documento crea files ed essi non possono non esistere, l'eccezione di file inesistente non è gestita in quanto l'idea di base che ho seguito impedisce la richiesta di un file inesistente.

3.1 Inizializzazione

Fase preliminare dove il server gestisce tutti i costruttori delle strutture dati e variabili in generale usate dal server. In particolare troviamo l'inizializzazione dello stub per l'RMI, usato per la fase di registrazione di un utente. Inoltre per la gestione degli inviti in diretta (live), viene attivato un Thread secondario che anche lui si comporterà da Listener, semplicemente per offrire un altro servizio che vedremo più avanti. Infine vi è un ThreadPool Executor che gestirà le richieste dei clients. Quest'ultimo è un FixedThreadPool e le motivazioni del perché sono varie: - ho evitato la scelta del CachedThreadpool in quanto la macchina ospite del progetto dovrà far girare molti processi (lato server e lato client) e reputo sia meglio poter contenere tale numero grazie ad un indicatore specificabile in un file di configurazione. - la FixedThreadPool è indicata in caso di CPU intensive tasks ed essendo le connessioni persistenti (come specificherò più avanti) trovo corretta questa scelta, soprattutto considerando che non avrei un vero e proprio riuso di tali Threads.

3.2 Specializzazione

Una volta finita la parte d'inizializzazione, ho optato per rendere il Server il Listener stesso del progetto senza avviare un ulteriore Thread. Esso quindi si mette in un ciclo eterno dove attenderà sulla welcomeSocket (letteralmente, la socket di "benvenuto") le richieste di connessione da parte dei vari clients: ogni connessione, una volta accettata, verrà poi passata al ThreadPool Executor in modo da non bloccare la possibilità ad ulteriori clients di connettersi.

3.3 Gestione richieste utente (RequestHandler.java)

Ogni volta che una connessione è accettata, viene creato un Runnable eseguito dal ThreadPool Executor. Tale Runnable è il RequestHandler, che, come suggerisce il nome, prenderà in carico le richieste da parte del Client connesso e le gestirà. Esso si mette in attesa in un ciclo "eterno" di un comando inviato dal Client (o della sua terminazione), per poi eseguire operazioni sulle strutture e restituire risultati. La modalità di domanda/risposta appena citata, viene gestita tramite un semplice protocollo TCP: abbiamo una Socket che connette il Client ed il

RequestHandler e, da questa, si ottiene un BufferedReader su cui ricevere la richiesta del client ed un DataOutputStream su cui scrivere il risultato della richiesta. Cosa gestisce il RequestHandler credo sia meglio se venga descritto durante la spiegazione dell'interfaccia del Client, quindi rimando tale parte. Il motivo per cui prima ho specificato con le virgolette "ciclo eterno" è che il Client ovviamente può disconnettersi (volontariamente con il chiudere l'interfaccia o meno per via di un crash e simili) ed in tal caso il RequestHandler gestisce il tutto in modo da cercare di mantenere intatto il database interno: per esempio, disconnette forzatamente l'utente che stava gestendo, libera la sezione di documento che stava modificando (se ne stava modificando una), chiude la Socket e termina seguendo il più possibile l'idea di un Graceful Shutdown.
Questo è un buon momento per specificare una scelta progettuale importante: un client ed un RequestHandler sono collegati da una connessione persistente per tutta la durata della vita del Client. Quando la Socket viene chiusa è necessariamente una scelta di uscita da parte dell'utente o un caso di errore sollevato.

3.4 Inviti Live: (PendingInvites.java)

Come il server principale, è un thread che fa da Listener. Attende su una welcomeSocket connessioni che avverrano ogni volta che un utente si connette al servizio e, sempre come il server, gestisce con un ThreadPoolExecutor le richieste attivando dei Runnables (PendingInviteHandler.java)

3.5 Gestione Inviti Live: (PendingInviteHandler.java)

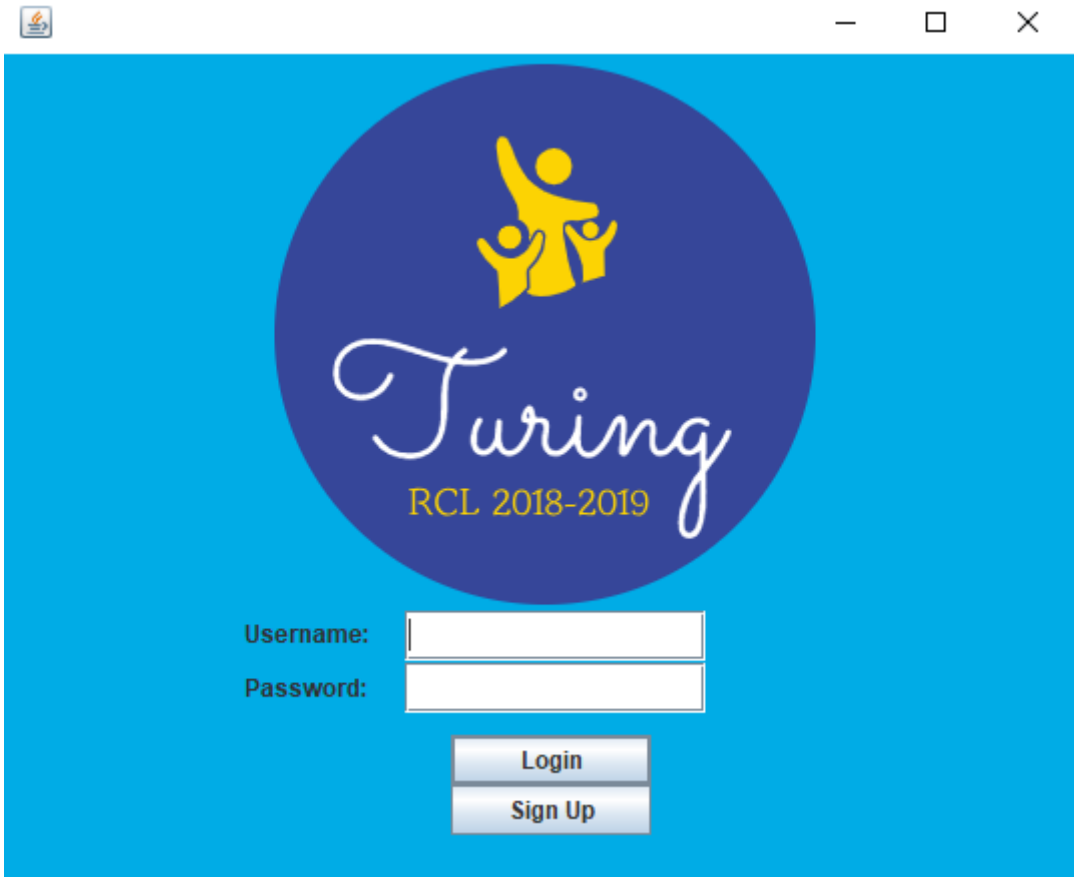
Una volta che è partito uno di questi Runnables, si mette in attesa di ricevere il nome utente da servire: questo passo è necessario affinché il Runnable sappia quali inviti, riferiti a quale utente, monitorare. Esso si mette in un "ciclo eterno" (come prima) dove semplicemente invia costantemente ciò che trova in un Set.

4 Lato Client

Entriamo ora nei dettagli di cosa accade quando si utilizza il Client. Per semplicità e coerenza con lo stesso scopo del progetto, seguiremo un classico flusso d'esecuzione.

4.1 Registrazione e Login (GUIClass.java)

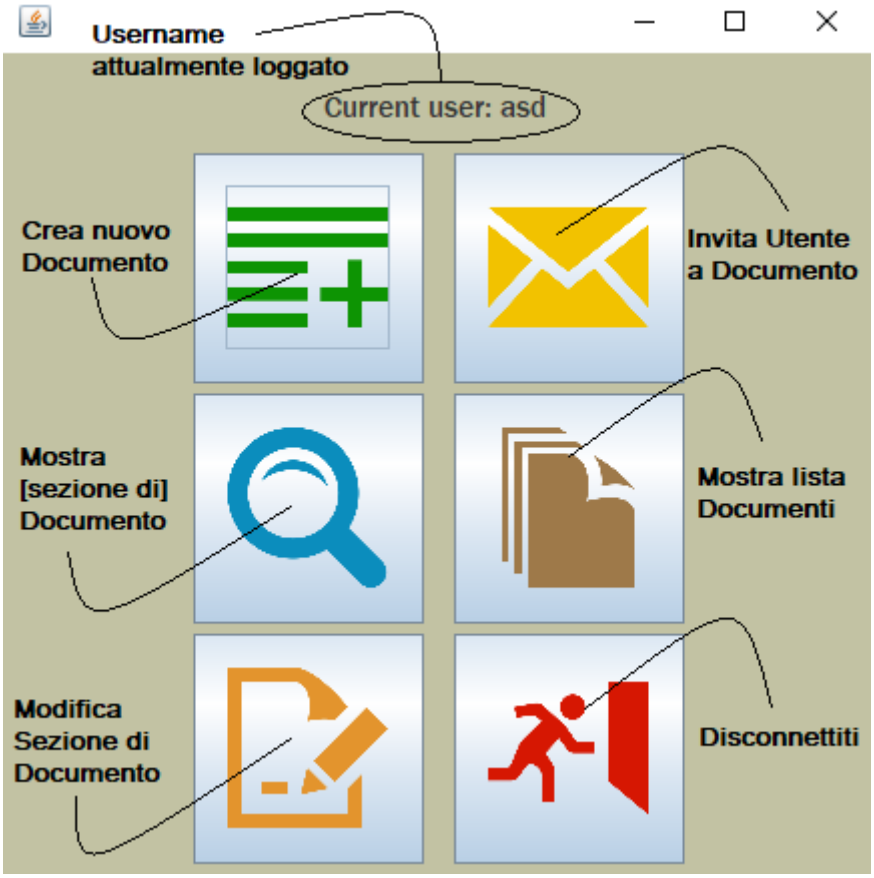
Una volta avviato ci troveremo di fronte questa schermata:



Se il server non dovesse essere attivo, verrà un semplice pop-up che ci informerà della situazione. In caso contrario, basterà scrivere, come suggerito, un nome utente ed una password per potersi registrare e successivamente loggare. Ripeto che non vi è persistenza dei dati tra un'esecuzione e l'altra del server, quindi qualsiasi lavoro svolto durante una sessione viene perso quando il server viene fatto ripartire: non c'è da aspettarsi quindi che un utente registrato nella sessione precedente sia presente in quella nuova, a meno di registrarlo nuovamente!

4.2 Operazioni di gestione (GUILoggedClass.java)

Una volta che avremo eseguito un Login con successo, si presenterà la seguente schermata. La grafica è volutamente semplicistica in quanto non è oggetto del corso, ma ho cercato, nella sua semplicità, di mantenere uno stile d'impatto, usando dei bottoni che richiamino la loro funzionalità. L'applicazione, ovviamente, non riporta le scritte in grassetto, usate qui solo come ulteriore indicazione.
Specifico che ogni finestra di supporto che verrà discussa esegue semplici controlli sugli inputs, per esempio che i nomi utenti siano per forza composti di lettere e/o numeri (unici validi caratteri anche per la registrazione), o che il numero di sezioni sia inferiore al numero massimo di sezioni di sistema, oppure che il numero di sezioni sia effettivamente un numero e così via. Non discuterò riguardo a come sono realizzate certe cose, qua verrà data una semplice panoramica del progetto.



4.2.1 Username Label

Notiamo in alto la presenza del nostro Username. Esso non è un semplice fatto stilistico, ma è utile in quanto, quando verranno aperti più clients, può essere complicato ricordarsi quale client è di chi, quindi funge da punto di

riferimento.

4.2.2 CreateDoc Button

Il primo pulsante che notiamo, seguendo la normale disposizione degli elementi (dall'alto verso il basso, da sinistra verso destra) è in effetti la base da cui si parte per l'utilizzo di Turing, ovvero la creazione di un Documento. Una volta premuto, si aprirà una piccola finestra che chiederà in input i due dati "decidibili" dall'Utente riguardo il documento, ovvero il nome (identificatore) ed il numero di sezioni. Tale azione è irreversibile, in quanto non c'è alcuna operazione di modifica o di elimina documento.

4.2.3 Invite Button

Il secondo pulsante, inerente all'invitare un utente ad un documento, serve in quanto ogni documento è riservato al suo creatore e agli utenti che quest'ultimo ha deciso, appunto, di invitare. Anche qui, premendo tale pulsante si aprirà una piccola finestra che richiederà due inputs, ovvero il nome del Documento a cui vogliamo invitare e l'utente da invitare.

4.2.4 Show Button

Il terzo pulsante, la lente d'ingrandimento, permette di visualizzare un documento o parte di esso. Per via di alcune complicanze nella gestione dell'input inerente alla sezione da visualizzare, ho scelto di procedere così:

- Se la sezione richiesta è minore del numero massimo di sezioni possibili, esso tenta di visualizzare il file richiesto.
- Se la sezione richiesta è maggior del numero massimo di sezioni possibili, verrà visualizzata una finestra d'errore.
- Se invece la sezione è uguale al numero massimo di sezioni possibili (che non può essere quindi una sezione del file effettivo), esso tenta di visualizzare il documento intero.

Il tutto è ovviamente specificato in breve anche nell'interfaccia come piccolo reminder della scelta implementativa. Il file viene scaricato in una cartella Downloads/nomeutente/fileNUMSEZ.txt se parziale o file_COMPLETE.txt se intero.

4.2.5 List Button

Il quarto pulsante permette la visualizzazione della lista dei documenti che un utente è abilitato a modificare, ovvero i documenti di cui egli stesso è creatore o per i quali è stato precedentemente invitato, quindi collaboratore. Ho optato per una finestra univoca riassuntiva del documento invece di una finestra per ogni documento. Questa scelta comporta che se un utente è invitato e/o creatore di troppi documenti, tale schermata diventi molto lunga. Per via della non persistenza dei dati e della piccolezza della questione, oltre al vantaggio dell'avere una sola finestra, ho optato per questa soluzione invece dell'altra, che comunque rischia di ottenere lo stesso risultato con una lista di collaboratori molto lunga!

4.2.6 Logout Button

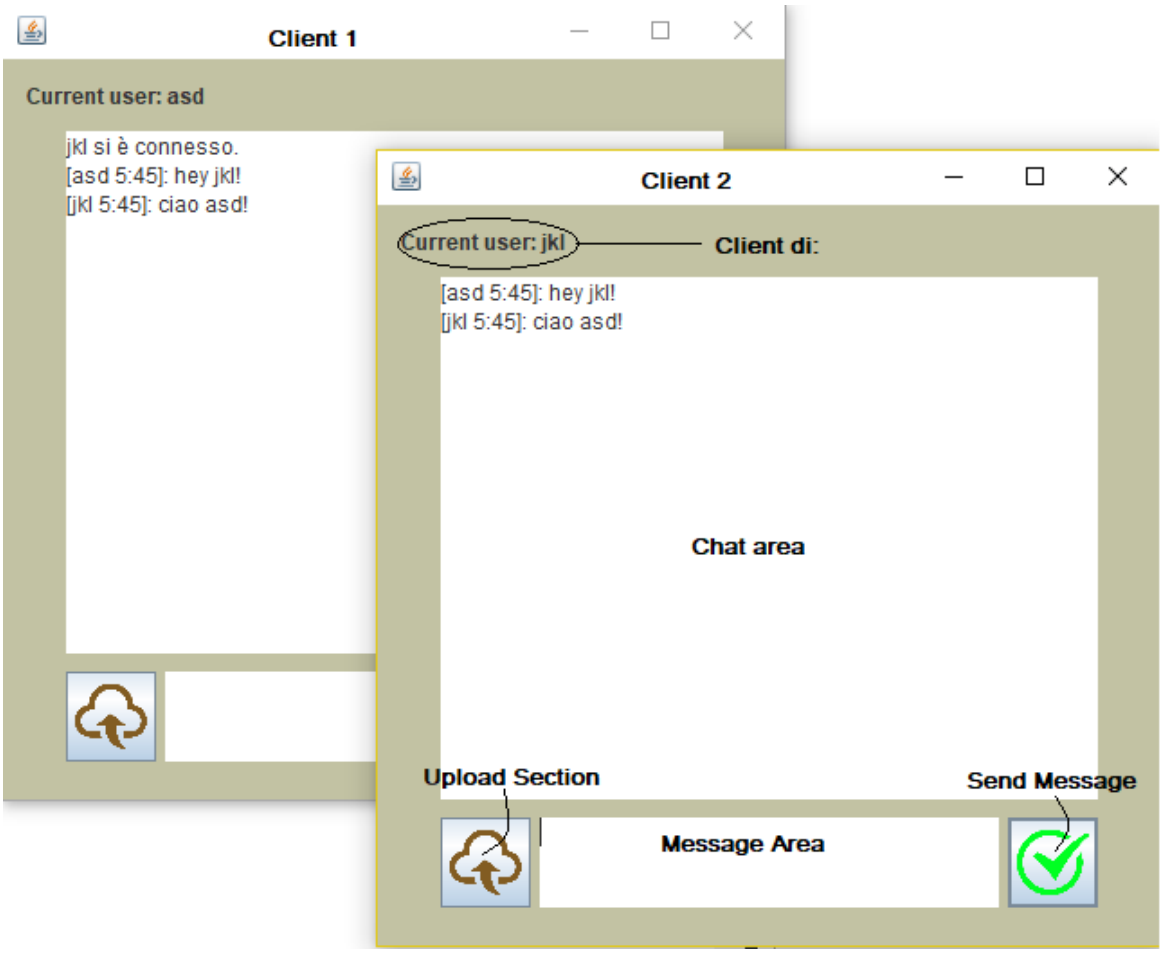
Anticipo la discussione di questo pulsante, il sesto, in modo da avere una spiegazione meno spezzettata riguardo il prossimo. Esso permette semplicemente di tornare alla finestra di partenza dove è possibile fare nuovamente Registrazione e/o Login.

4.2.7 Edit Button

Il punto focale di Turing viene abilitato da questo pulsante. Esso permette di specificare il nome del Documento e la Sezione che si desidera modificare. Se la richiesta va a buon fine, otteniamo che l'interfaccia cambia e passiamo in modalità di Editing, che a breve descriveremo. Il file viene scaricato nella cartella Editing/nomeutente/file#.txt.

4.3 Operazioni di Modifica (GUIEditClass.java)

Una volta entrati in modalità modifica, ci si connetterà ad una chat con tutti quelli che stanno già modificando una sezione del documento scelto. Vedasi l'immagine per avere un'idea generale:



Notiamo la presenza di due nuovi bottoni, uno inerente all'inviare il messaggio che un utente voglia ipoteticamente scrivere sulla chat (di default viene catturato l'evento del premere la Enter key) ed uno che permette di finire la modalità di editing e tornare alla schermata di gestione precedente (GUILoggedClass). In questo caso, in quanto il file da modificare è stato precedentemente assegnato e scaricato, non è necessario che il bottone Upload Section richieda in input i parametri del file, in quanto alla fine dell'Editing dovrebbe essere stata modificata solo la propria sezione e quindi dovrebbe essere locata dove è stata precedentemente scaricata.

Part III
Implementazioni

Passiamo ora ad una spiegazione di come certe cose sono state implementate: le "porte note" e concetti simili si riferiscono a valori specificati in un file di configurazione, Configurations.java.

5 Schermata di Login

Registrazione

VEDI SLIDES RMI

5.1 Login

Implementato tramite una normale connessione TCP, è il metodo prevalentemente usato nelle modalità di richiesta/risposta. Una Socket, ovviamente su localhost e con porta nota, viene utilizzata ottenendo i relativi DataOutputStream e BufferedReader per, rispettivamente, la scrittura di richieste e la lettura di risposte verso/da il server. Una volta che la richiesta ha avuto successo, l'username viene rimosso dal Set degli utenti offline ed aggiunto al Set degli utenti online. Inoltre, il Client cerca di ottenere subito le informazioni inerenti agli inviti che ha ricevuto mentre l'utente era offline e fa partire un Listener per la ricezione di inviti in diretta (vedasi la sezione Supporto poco sotto).

6 Schermata di Gestione

6.1 Crea Documento

Come nel login, è gestito tramite una connessione TCP verso il server dove vengono inviati i vari campi necessari. Il Request Handler che gestisce il client richiama un metodo del server che restituisce l'esito della richiesta, per poi inoltrarla al client che intanto si è messo in attesa della risposta. Se l'esito è di successo, otteniamo come risultato che nella cartella Documents (relativa solo al server) vi è una cartella con il nome del documento con all'interno X files di formato .txt vuoti, identificati come nomeDoc0.txt, nomeDoc1.txt, ... nomeDocX-1.txt .

6.2 Invita a Documento

Gestione TCP identica alla precedente. Se l'esito è di successo, otteniamo due comportamenti distinti a seconda se l'utente è online o offline:

- se l'utente è offline, viene inserito il nome del documento in un Set privato dell'utente che verrà richiesto in fase di login;
- se l'utente è online, viene inserito il nome del documento in un Set privato dell'utente che verrà immediatamente spedito dal Thread di competenza (vedasi la sezione Supporto).

Come risultato, a prescindere dallo status dell'utente, è che l'utente invitato è ora nella lista degli utenti abilitati alla modifica del Documento e l'utente ha nel proprio insieme di Documenti il nome del documento. È un doppio controllo con ridondanza delle informazioni che permette però di avere accesso, sia da parte Utente sia da parte Documento, a tali informazioni, senza interrogare necessariamente l'intero database.

6.3 Mostra [Sezione di] Documento

Gestione TCP identica alla precedente. In caso di esito positivo, come specificato durante la presentazione dell'interfaccia grafica, a seconda del numero di sezione specificato si ottiene l'intero documento o la sezione richiesta. Il documento, che sia intero o meno, passa prima da un FileChannel per la sua lettura, ad un SocketChannel a cui viene inviato verso il client richiedente, per poi tornare ad un FileChannel per la sua scrittura nella destinazione di default (Downloads/user/nomeDoc): in quanto era richiesto che la gestione dei files fosse NIO, troviamo quindi ByteBuffers e SocketChannels. Nel caso in cui il file debba essere intero semplicemente ci saranno più FileChannels di lettura.

6.4 Lista Documenti

Gestione TCP identica alla precedente. In caso di esito positivo, viene richiesto il Set dei documenti che l'utente è abilitato a modificare (ecco il risultato della ridondanza delle informazioni precedenti) e semplicemente scritto su una finestra di supporto.

6.5 Modifica Sezione di Documento

Gestione TCP identica alla precedente per la fase di richiesta. In caso di esito positivo, viene scaricato il file seguendo una linea estremamente simile a quella spiegata nel comando Mostra [Sezione di] Documento, in particolare quella non del file intero. Il file viene scaricato nella destinazione di default (Editing/user/file).

6.6 Logout

Gestione TCP identica alla precedente. Non ho trovato modi di rendere l'esito della richiesta non positivo, ma di nuovo, in caso di esito positivo il nome utente viene rimosso dal Set degli utenti online ed aggiunto al Set degli utenti offline (procedimento inverso del Login).

7 Schermata di Editing

7.1 Invia Messaggio

La chat è implementata mediante Multicast UDP. Un utente entra a far parte del gruppo d'interesse del documento non appena entra in fase di editing e smette di esserlo non appena ne esce, in modo da limitare l'invio del messaggio ai soli utenti interessati, ovvero a quelli che effettuano modifiche nello stesso momento di altri.

7.2 Upload File

Procedimento identico, seppur inverso, del "Modifica Sezione di Documento" precedente. FileChannel per leggere il file (client), SocketChannel (con ByteBuffer) per inviare il file al server, FileChannel per salvare il file (server):

classica gestione NIO.

8 Supporto

8.1 Server: Sender di Inviti Live

In quanto deve esistere la possibilità di ricevere una notifica live di un ipotetico invito, ho optato per l'utilizzo di un Runnable che richiede costantemente un determinato Set del nome utente che sta servendo. Gli inviti verso utenti online, come abbiamo visto precedentemente, utilizzano un Set a parte rispetto agli inviti verso utenti offline, e questo Runnable cerca aggiornamenti in tale Set per inviarli tramite TCP al Client di riferimento. Il client si connette tramite una welcomeSocket ed una porta nota al Listener a riguardo per smistare e gestire le richieste (PendingInvites.java) e fa partire Runnables (PendingInviteHandler.java) che gestiranno tale loop fino alla disconnessione dell'utente.

8.2 Client: Listener per Inviti Live

Siccome esiste un Sender di inviti, dovrà esistere anche qualcosa lato client pronto a ricevere tali notizie: ecco qui il Listener, ironicamente chiamato NotSoGUIListener, in quanto non ha un'interfaccia grafica ma serve affinché questa possa avere il servizio di invite live.

8.3 Client: Listener per Chat

Infine per quanto riguarda la chat in fase di Editing esiste un Thread di supporto (Chat.java) che controlla l'esistenza di nuovi messaggi e, se ne trova, li appende all'area di Chat dell'interfaccia utente che ha creato tale Thread. Ovviamente, essendo UDP, troviamo i DatagramPackets. Esso viene terminato solo quando si lascia la fase di editing, sempre seguendo la logica del Graceful Shutdown, semplicemente settando a falso la guardia del suo ciclo. In breve, questo è lo Sniffer affrontato durante il corso di Laboratorio.

Part IV
Conclusioni

È stato sicuramente un progetto che mi ha divertito durante la sua creazione ed estremamente utile ai fini dell'apprendimento. Sono felice del risultato ottenuto e sono contento di aver potuto finalmente testare un'interfaccia grafica per un programma, era da tempo che desideravo farlo. Ho apprezzato il tentativo di immettere più argomenti possibili tra quelli affrontati durante il corso, rimanendo comunque in un ambiente coerente con l'idea di partenza e senza sovraccaricare il progetto: di nota sempre positiva oltretutto ho trovato la totale libertà nella gestione di vari campi, proprio come la questione dell'interfaccia grafica, la persistenza dei dati e così via.

Ringrazio i docenti per la loro disponibilità durante tutto il periodo del corso ed anche per questo periodo d'esami.

Giulio Purgatorio (516292)