



UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

RETI DI CALCOLATORI E LABORATORIO

2018/2019

Turing

Autore: Stefano Torneo

Docente: Federica Paganelli

Matricola: 545261

Corso: B

5 Febbraio 2019

Indice

1.	OBIETTIVO	3
2.	STRUTTURE DATI	3
2.1	<i>ConcurrentHashMap per gli utenti registrati</i>	3
2.1.1	<i>Gestione Concorrenza</i>	3
2.2	<i>ConcurrentHashMap per gli utenti online</i>	3
2.2.1	<i>Gestione Concorrenza</i>	3
2.3	<i>ConcurrentHashMap per i documenti</i>	4
2.3.1	<i>Gestione concorrenza</i>	4
2.4	<i>ConcurrentHashMap per i gruppi</i>	5
2.4.1	<i>Gestione concorrenza</i>	5
2.5	<i>ArrayList per gli indirizzi dei gruppi</i>	5
2.5.1	<i>Gestione concorrenza</i>	5
3.	ORGANIZZAZIONE DEI FILES	5
4.	STRUTTURA DEL CODICE	6
5.	STRUTTURA DEL SERVER	8
6.	STRUTTURA DEL CLIENT	8
6.1	PROTOCOLLI DI TERMINAZIONE	9
7.	MANUALE DI ISTRUZIONI	9
8.	RISORSE UTILIZZATE E COMPILAZIONE	12

1. Obiettivo

L'obiettivo del progetto è quello di realizzare un'applicazione client-server dove i client possono creare e modificare un insieme di documenti e condividere la modifica con altri utenti invitati.

2. Strutture dati

Le strutture dati utilizzate dal lato server sono:

- ConcurrentHashMap per gli utenti registrati.
- ConcurrentHashMap per gli utenti online.
- ConcurrentHashMap per i documenti.
- ConcurrentHashMap per i gruppi.
- ArrayList per mantenere traccia degli indirizzi assegnati ai gruppi.

2.1 ConcurrentHashMap per gli utenti registrati

Ho scelto di utilizzare una ConcurrentHashMap per la gestione degli utenti registrati, dove la chiave è l'username dell'utente e il valore è un oggetto della classe RegisteredUser.

Ogni utente registrato verrà inserito nella tabella hash, memorizzandone gli attributi della classe RegisteredUser, ovvero:

- password
- lista degli inviti ricevuti (un ArrayList di oggetti di tipo String)
- lista di documenti a cui collabora (un ArrayList di oggetti di tipo Documento)

2.1.1 Gestione Concorrenza

Usando una ConcurrentHashMap l'accesso ai dati è immediato, e mantiene un buon grado di concorrenza.

La lista degli inviti e quella delle collaborazioni sono istanziate come synchronizedList in modo che le invocazioni delle singole operazioni della collezione siano thread-safe, mentre per più metodi invocati insieme utilizzo le lock implicite (synchronized) sulla lista.

Quando accedo in lettura ad una lista, prendo la mutua-esclusione sulla lista in modo tale che lo stato sia consistente, perché un altro thread potrebbe modificare la lista mentre l'utente sta accedendo in lettura a quella lista.

2.2 ConcurrentHashMap per gli utenti online

Ho scelto di utilizzare una ConcurrentHashMap per la gestione degli utenti online, dove la chiave è l'username dell'utente e il valore è un oggetto della classe OnlineUser.

Ogni utente online verrà inserito nella tabella hash, memorizzandone gli attributi della classe OnlineUser, ovvero:

- socket delle notifiche
- ObjectOutputStream per l'invio delle notifiche

2.2.1 Gestione Concorrenza

Usando una `ConcurrentHashMap` l'accesso ai dati è immediato, e mantiene un buon grado di concorrenza.

Per ogni utente viene memorizzato un `ObjectOutputStream` per poter inviare le notifiche al thread Notifiche in ascolto per quell'utente.

Per evitare che due inviti o un invito e un messaggio di terminazione, inviati allo stesso utente contemporaneamente, si perdano, ho ritenuto opportuno eseguire in mutua-esclusione l'invio di notifiche, utilizzando una `ReentrantLock` dedicata ad ogni utente.

Quando si deve inviare una notifica ad un utente, si prende la mutua-esclusione sullo stream di quell'utente, si invia la notifica e dopo si rilascia la mutua-esclusione.

2.3 ConcurrentHashMap per i documenti

Ho scelto di utilizzare una `ConcurrentHashMap` per la gestione dei documenti, dove la chiave è il nome del documento (formato da nome del documento e nome del creatore) e il valore è un oggetto della classe `Documento`.

Ogni documento creato verrà inserito nella tabella hash, memorizzandone gli attributi della classe `Documento`, ovvero:

- nome
- creatore
- lista di sezioni (un `ArrayList` di oggetti di tipo `Sezione`)
- numero di sezioni
- numero di sezioni in editing
- lista di collaboratori (un `ArrayList` di oggetti di tipo `String`)

2.3.1 Gestione concorrenza

Usando una `ConcurrentHashMap` l'accesso ai dati è immediato, e mantiene un buon grado di concorrenza.

Inoltre, è stato necessario avere altre due variabili di mutua-esclusione per la lista dei collaboratori e per la lista delle sezioni.

Per la lista dei collaboratori ho ritenuto opportuno usare una `ReadWriteLock`, in quanto permette la modifica della lista ad un solo thread, ma la lettura a più thread, ovvero a più utenti che fanno richiesta di visualizzazione dei documenti a cui collaborano.

Per la lista delle sezioni, invece, ho ritenuto opportuno usare un array di `ReentrantLock` di lunghezza pari al numero di sezioni di quel documento. Quando un utente prova a modificare una sezione, se la sezione è libera allora prende la mutua-esclusione su quella sezione e la rilascia quando fa la end-edit oppure quando scade il timer di quella sezione o quello del client stesso, altrimenti se è già in fase di editing restituisce il controllo al chiamante senza bloccarsi, grazie all'utilizzo del metodo `tryLock()`. Inoltre, ogni sezione ha una propria `ReadWriteLock` per accedere in mutua-esclusione alle variabili utente (colui che sta modificando) ed editata (un booleano che indica se la sezione è in editing o meno). La scelta della `ReadWriteLock` permette a più utenti di accedere in lettura alle due variabili, mentre ad un solo utente di poterla modificare.

Ho utilizzato anche una `ReentrantLock` per modificare in mutua-esclusione la variabile `inEditing` presente nella classe `Documento`, che indica il numero di sezioni in editing, in modo da capire quando non ci sono più utenti che stanno modificando il documento e quindi che si può rilasciare l'indirizzo della chat associato a quel gruppo.

2.4 ConcurrentHashMap per i gruppi

Ho scelto di utilizzare una ConcurrentHashMap per la gestione dei gruppi, dove la chiave è il nome del gruppo (uguale al nome del documento corrispondente) e il valore è un oggetto della classe Gruppo. Ogni gruppo creato verrà inserito nella tabella hash, memorizzandone gli attributi della classe Gruppo, ovvero:

- indirizzo
- porta

2.4.1 Gestione concorrenza

Usando una ConcurrentHashMap l'accesso ai dati è immediato, e mantiene un buon grado di concorrenza.

Politiche Gruppi:

1. Viene creato un gruppo per ogni documento e gli viene assegnato un indirizzo e una porta solo nel momento in cui un utente inizia a modificare una sezione di un documento di cui non ci sono altre sezioni già in fase di editing.
2. Se c'è un solo utente che sta editando il documento e questo chiude la sezione, allora l'indirizzo assegnato al gruppo viene rimosso dalla lista degli indirizzi utilizzati e può essere assegnato ad un altro gruppo.
3. Numero massimo di gruppi pari a 254 ovvero numero massimo di indirizzi disponibili, quando non ci sono più indirizzi disponibili, l'utente può effettuare l'editing della sezione ma non utilizzare una chat per comunicare con gli altri collaboratori.
4. Per far terminare la chat di un membro di un gruppo, ovvero il thread in ascolto sui messaggi inviati sul gruppo, ho deciso che l'utente che vuole abbandonare la chat deve inviare un messaggio di terminazione formato da *username-quit*.

2.5 ArrayList per gli indirizzi dei gruppi

Ho scelto di utilizzare una ArrayList istanziata come synchronizedList per mantenere traccia degli indirizzi utilizzati dai vari gruppi.

Quando un utente effettua l'editing di una sezione e ogni altra sezione è libera, allora si inserisce un indirizzo generato casualmente, nel range 224.1.1.1 – 224.1.1.254, nella struttura degli indirizzi.

Quando non ci sono più sezioni in editing per un documento, allora l'indirizzo assegnato ai collaboratori di quel documento viene rimosso dalla lista degli indirizzi utilizzati.

2.5.1 Gestione concorrenza

Usando una synchronizedList le invocazioni delle singole operazioni della collezione sono thread-safe, mentre per più metodi invocati insieme utilizzo le lock implicite (synchronized) sulla lista. L'utilizzo della mutua-esclusione per questa struttura mi garantisce il fatto che due utenti che accedono alla modifica di una sezione (di documenti diversi) nello stesso istante non ricevono lo stesso indirizzo.

3. Organizzazione dei files

Per ogni documento creato, vengono creati n files dove n è il numero di sezioni. Il nome dei files è formato dall'unione di *nomedocumento* e *numero sezione*.

Il nome di un documento è reso univoco, aggiungendo il nome del creatore al nome del documento scelto dal creatore stesso. In questo modo ci possono essere documenti con lo stesso nome di creatori diversi, ma i due documenti sono considerati diversi.

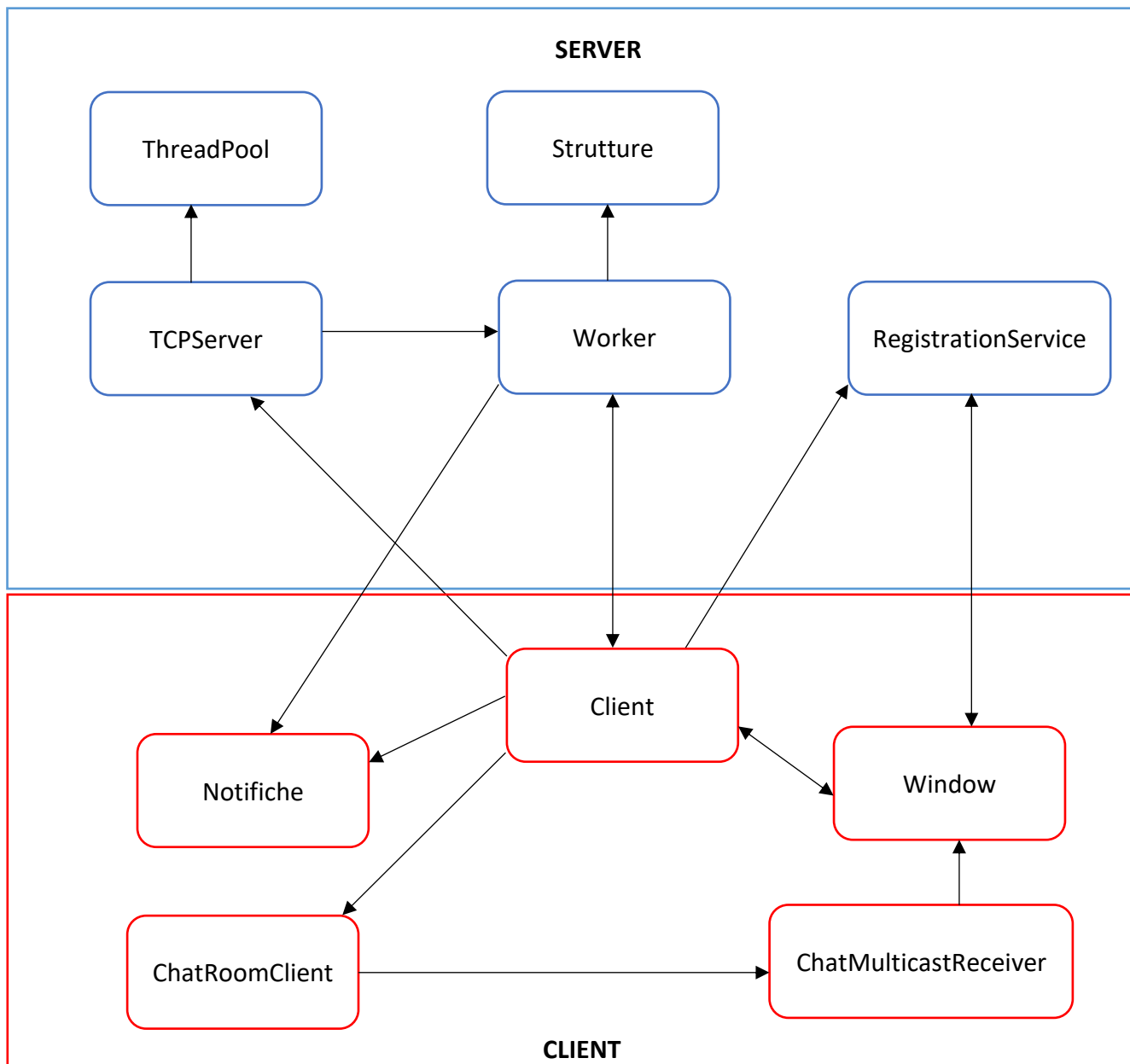
Questo è stato possibile sfruttando l'univocità dell'username.

Inoltre, viene creata una cartella (con nome uguale all'username) per ogni utente, in modo tale che i files creati da un utente stiano nella stessa cartella e siano facilmente reperibili.

4. Struttura del codice

Il codice è stato diviso varie classi, tra le quali:

- **Server**: classe principale del server dove vengono creati il servizio RMI, il threadPool per la gestione dei worker e il server TCP.
- **TCPServer**: classe che si occupa principalmente di ascoltare le richieste dei vari client, e quando arriva una richiesta di connessione crea un socket per le richieste del client e un socket per le notifiche di quel client, crea un nuovo oggetto della classe Worker e gli passa le due socket create, dopodiché passa il thread al threadPool in modo che possa eseguirlo.
- **Client**: classe principale del client, inizialmente si occupa di instaurare una connessione con il server tramite RMI, dopodiché invia richieste e riceve risposte dal server.
- **Window**: classe che gestisce tutta la parte grafica dell'applicazione, ogni client ha una propria istanza della classe Window in modo che ogni client abbia una propria interfaccia con la quale comunicare con il server.
- **Worker**: classe in cui sono presenti le implementazioni di tutte le funzionalità che si possono fare sui documenti. Rappresenta la classe eseguita dal thread in attesa di richieste da parte di un client specifico, quando gli arriva una richiesta, cerca di soddisfarla e invia l'esito al client corrispondente.
- **ChatMulticastReceiver**: classe utilizzata per ascoltare i messaggi che arrivano in un gruppo.
- **ChatRoomClient**: classe utilizzata per permettere ad un client di inviare un messaggio nella chat del gruppo a cui appartiene in quel momento.
- **RegistrationService**: classe che rappresenta l'interfaccia dei metodi utilizzati dalla classe RegistryServiceImpl.
- **RegistrationServiceImpl**: classe che implementa i metodi dell'interfaccia RegistryService. Il metodo che implementa è quello di registrazione dell'utente nella struttura degli utenti registrati.
- **ThreadPool**: classe che crea un threadPool per l'esecuzione dei vari task passati. Ogni task rappresenta un oggetto della classe Worker.
- **Notifiche**: classe utilizzata per la ricezione delle notifiche che vengono inviate ad un utente.
- **RegisteredUser**: classe che definisce gli attributi memorizzati per ogni utente registrato.
- **OnlineUser**: classe che definisce gli attributi memorizzati per ogni utente online.
- **Documento**: classe che definisce gli attributi memorizzati per ogni documento e un metodo per la creazione degli n files che rappresentano le n sezioni del documento.
- **Sezione**: classe che definisce gli attributi memorizzati per ogni sezione di un documento.
- **Gruppo**: classe che definisce gli attributi memorizzati per ogni gruppo.
- **Richiesta**: classe che definisce i metodi e gli attributi per inviare una richiesta e/o una risposta tra client e server.
- **Strutture**: classe che definisce le strutture utilizzate dal server per immagazzinare le informazioni su utenti, documenti, gruppi e indirizzi.



Comunicazione tra le varie classi:

- **Client->Window:** viene creato un oggetto di tipo Window per la visualizzazione delle varie finestre.
- **Window->Client:** vengono richiamati i metodi della classe Client, in base alle richieste dell'utente tramite interfaccia.
- **Client->ChatRoomClient:** creazione di una nuova chat, invio di messaggi, terminazione thread per la chat.
- **ChatRoomClient->ChatMulticastReceiver:** viene creato un thread che ascolta i messaggi della chat del gruppo.
- **ChatMulticastReceiver->Window:** il thread in ascolto sui messaggi della chat, appende i messaggi nell'area di testo passatogli come argomento.
- **Client->RegistrationService:** il client si connette al server, che si occupa delle registrazioni, tramite RMI.
- **Window->RegistrationService:** il client invia una richiesta di registrazione al server tramite RMI.

- **RegistrationService->Window**: il server tramite RMI invia l'esito della registrazione (successo o altro) e viene stampato un messaggio corrispondente.
- **Client->Notifiche**: quando il client viene loggato, crea un thread per l'ascolto delle notifiche.
- **Client->TCPServer**: il client si connette al server tramite indirizzo e porta.
- **TCPServer->Worker**: il server quando riceve una richiesta di connessione, crea un nuovo oggetto della classe Worker.
- **Client->Worker**: il client invia una richiesta al thread del server dedicato a soddisfare le sue richieste.
- **Worker->Client**: il worker invia al client l'esito della richiesta.
- **Worker->Notifiche**: il worker invia dei messaggi al thread dedicato all'ascolto delle notifiche, per notificargli inviti o terminazione.
- **TCPServer->ThreadPool**: il server TCP quando riceve una richiesta, passa un task (un oggetto della classe Worker) al threadpool che pensa ad eseguirlo.
- **Worker->Strutture**: il worker interagisce con le varie strutture dati per soddisfare le richieste dei client.

5. Struttura del Server

Il server è strutturato in modo da avere:

- Un **servizio RMI**: per registrare gli utenti.
- Un **listener**: che ha il compito di ascoltare nuove connessioni, accettarle e creare un worker da passare al threadpool per essere eseguito.
Il listener è in attesa di due connessioni, la prima riferita alle richieste del client, la seconda riferita alle notifiche, in modo da creare due socket diverse per richieste e notifiche.
- E più **workers**, uno per ogni client che richiede la connessione.
Un worker ascolta le richieste che arrivano dal client per cui è stato creato, soddisfa la richiesta e dopo invia una risposta per indicargli l'esito della richiesta.
In attesa di richieste, un worker installa un timer, se non arriva nessuna richiesta e il time-out scatta allora la connessione con quel client viene interrotta.
Viene installato un timer anche ogni volta che un utente accede alla modifica di una sezione, quando scatta il time-out, la connessione client-server viene chiusa.

Il server ha un indirizzo ben definito "127.0.0.1" e due porte "49999" e "49998", la prima per ascoltare le richieste di connessione, la seconda per creare il socket delle notifiche.

6. Struttura del Client

Il client è strutturato in modo da avere:

- Un thread (main) per inviare richieste e ricevere risposte dal server tramite connessione TCP.
Si connette al server tramite indirizzo e porta conosciute, una volta che la richiesta di connessione è stata accettata, invia una richiesta per la creazione di un'altra socket da passare al thread delle notifiche. Inoltre, crea due stream, uno per la ricezione e uno per l'invio, in modo da poter comunicare con il server tramite TCP.
- Un thread per unirsi ad un gruppo ed ascoltare i messaggi che arrivano da quel gruppo.
- Un thread per rimanere in ascolto di notifiche. Viene creato uno stream di input sul socket delle notifiche. Le notifiche possono arrivare dal server quando un altro utente lo invita a collaborare ad un documento, oppure, quando il thread deve terminare, il server gli invia un messaggio di terminazione.

6.1 Protocolli di terminazione

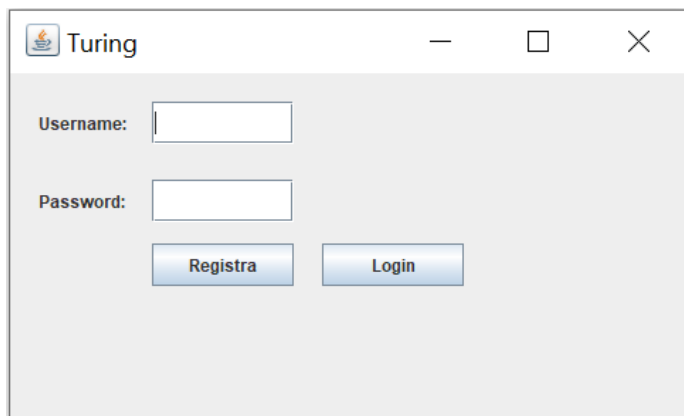
Per fare terminare i vari thread ho adottato diverse politiche:

- **Worker:** termina quando il client fa una richiesta di Logout, oppure quando scade il time-out relativo al client o ad una sezione. In tutti e tre i casi, il client viene rimosso dagli utenti online, vengono chiusi gli stream di input e output per la comunicazione TCP; inoltre nel caso di Logout viene inviato un messaggio di terminazione al thread Notifiche in ascolto, mentre in caso di time-out per la sezione viene rilasciata la lock della sezione prima di terminare in modo che possa essere utilizzata da altri client.
- **Notifiche:** termina quando rileva un'eccezione sullo stream di input (ciò indica per esempio che il server ha crashato) o quando riceve un messaggio di terminazione identificato da una richiesta con tipo "-2".
- **Chat:** termina quando viene inviato un messaggio speciale di terminazione sulla chat del gruppo. Il messaggio di terminazione speciale è *username-quit*, dove *username* indica il nome dell'utente che vuole abbandonare la chat. I thread in ascolto sui messaggi relativi ad un gruppo, quando ricevono un messaggio controllano se corrisponde al messaggio di terminazione, e se sì, verificano che l'username presente nel messaggio di terminazione coincida con l'username di chi ha mandato il messaggio in modo da evitare che un utente possa far terminare la chat di un altro utente.

7. Manuale di istruzioni

L'applicazione Client è divisa in due finestre principali:

FINESTRA DI REGISTRAZIONE E LOGIN

The image shows a window titled 'Turing' with a standard Windows-style title bar (minimize, maximize, close buttons). Inside the window, there are two text input fields. The first is labeled 'Username:' and the second is labeled 'Password:'. Below these fields are two buttons: 'Registra' (on the left) and 'Login' (on the right). The window has a light gray background.

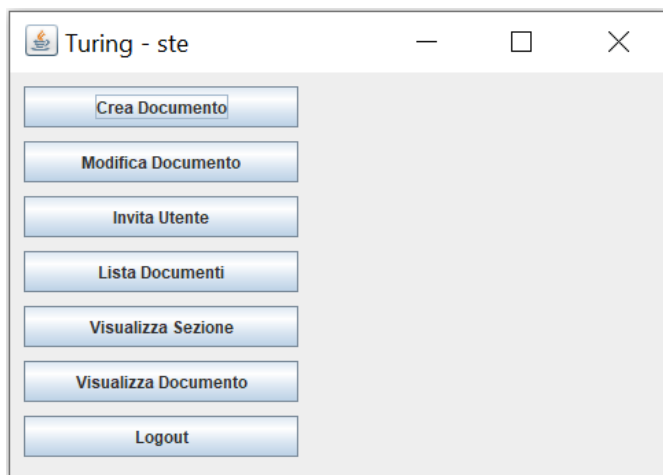
Per registrarsi:

- Inserire l'username
- Inserire la password
- Premere il bottone 'Registra'
- Verrà visualizzato un messaggio di errore o successo, in quest'ultimo caso vengono visualizzati username e password inseriti.

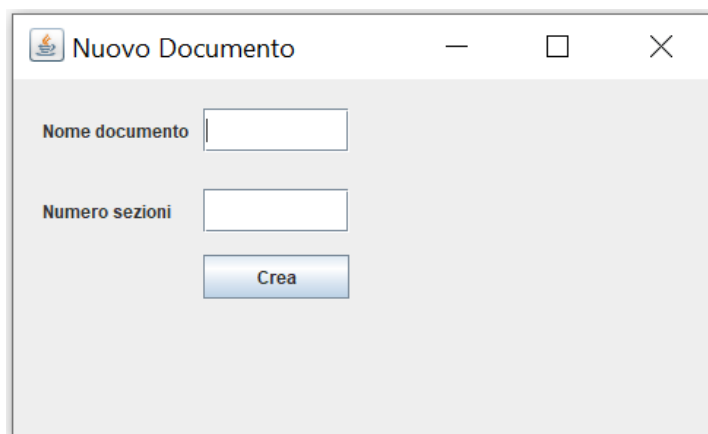
Per loggarsi:

- Inserire username
- Inserire password
- Premere il bottone 'Login'
- Verrà visualizzato un messaggio di errore o successo, in quest'ultimo caso viene aperta la finestra per utilizzare le varie funzionalità dell'applicazione.

FINESTRA CON LE VARIE FUNZIONALITA'



In questa finestra, in alto viene visualizzato l'username dell'utente.
Premendo, ad esempio, il bottone 'Crea Documento', si apre la seguente finestra:



Per creare un nuovo documento:

- Inserire il nome del documento
- Inserire il numero di sezioni
- Premere sul bottone 'Crea'
- Verrà visualizzato un messaggio di errore o successo.

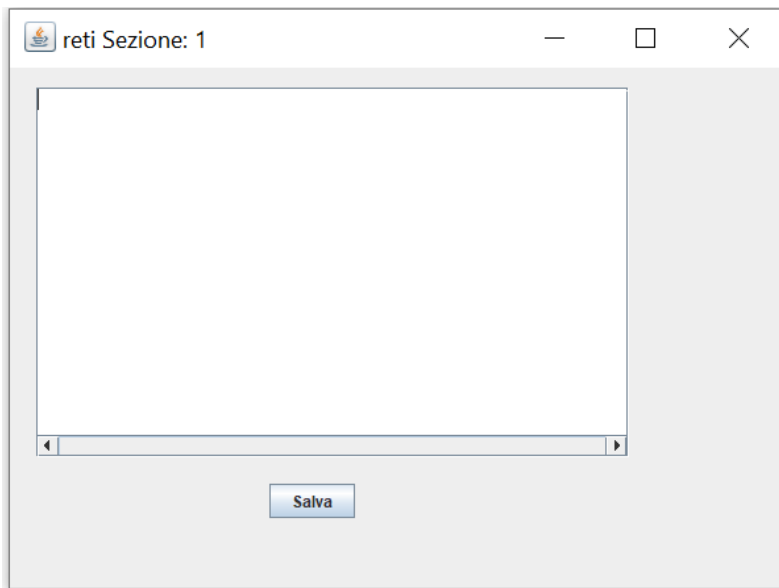
Le altre finestre che vengono visualizzate quando si preme sui seguenti bottoni:

- 'Modifica Documento'
- 'Visualizza Sezione'
- 'Visualizza Documento'

hanno un formato simile alla finestra per creare un nuovo documento.

Quando si va a modificare una sezione di un documento, dopo aver premuto 'Modifica Documento' e aver compilato i campi richiesti in modo corretto, si aprono due finestre:

- Una per modificare la sezione

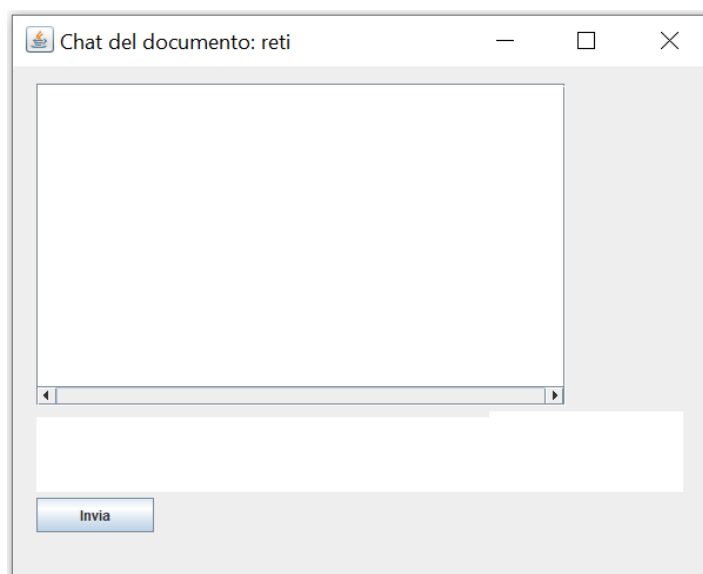


Dove in alto viene visualizzato il nome del documento e la sezione che si sta editando.

La finestra può essere chiusa in due modi:

- Premendo sul tasto 'Salva'
- Cliccando sull'apposito tasto di chiusura della finestra (X, in alto a destra), in tal caso le modifiche effettuate non vengono salvate.

- Un'altra per la chat

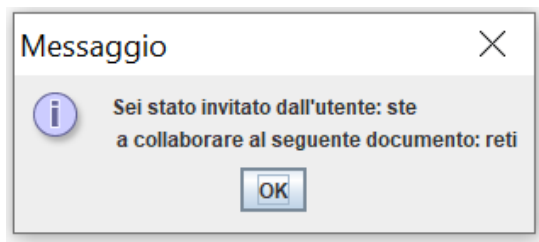


Dove in alto viene visualizzato il nome del documento.

La finestra è composta da:

- Un'area per visualizzare i messaggi inviati nel gruppo.
- Un'area per scrivere i messaggi da inviare nel gruppo.
- Un bottone 'Invia' per inviare i messaggi.

All'arrivo delle notifiche, viene visualizzato un messaggio di questo tipo:



Il Logout può essere effettuato in due modi:

- Premendo sul bottone 'Logout'.
- Cliccando sull'apposito tasto di chiusura della finestra (X, in alto a destra).

Tutte e due le azioni intraprese, porteranno l'utente alla finestra di Registrazione/Login.

8. Risorse utilizzate e compilazione

Il progetto è stato sviluppato in una macchina Windows 10 a 64 bit utilizzando il software Eclipse 4.9.0.

Per compilare i files da prompt bisogna eseguire i seguenti comandi:

- `javac Server.java`
- `javac Client.java`

Per compilarli da Eclipse, basta mandare in esecuzione Server e Client.