# Relazione progetto Reti di Calcolatori Laboratorio 2018/2019

**Autore: Samuele Intaschi**
**Matricola: 523864**
**Corso A**

## Presentazione

Il programma si occupa di gestire la produzione collaborativa di documenti.
Un utente può creare un documento di testo e successivamente invitare altri utenti iscritti a collaborare alla scrittura di questo. La modifica avviene per sezioni, cioè alla creazione viene specificato un numero di sezioni in cui è diviso il documento e l' utente ne può modificare una alla volta.
Il client, oltre a funzionare da linea di comando, implementa un' interfaccia grafica semplice.

## 1. Architettura server e threads utilizzati

Il programma opera tramite un' architettura tradizionale client-server, in cui il client invia richieste al server e questo affida il compito di gestirle ad un thread di un pool, che continua poi la comunicazione con il client al fine di portare a termine il servizio richiesto.
Il server per semplicità usa un pool di thread formato da ventiquattro thread, un numero non troppo basso ma tale da non introdurre troppo overhead per i cambi di contesto in caso di carico non estremo, come nel caso del test proposto insieme al progetto, ma potrebbe essere aggiunto un metodo per stabilire la quantità di carico a cui è sottoposto il server e di conseguenza adattare il numero di thread utilizzati.
Il thread listener del server opera in modalità non blocking, quindi con un *selector* accetta connessioni e stabilisce su quali di questi socket c'è una richiesta da leggere, per toglierli dall' insieme ed affidarli ad un thread worker del pool, che invece opera in modalità blocking per comunicare con il client.
I documenti, con le loro sezioni, e gli utenti sono mantenuti in memoria sempre aggiornati sotto forma di oggetti JSON e in questo formato vengono anche mandati al client i documenti in risposta alle richieste di modifica o visualizzazione.
Anche le richieste inviate dai client al server sono in formato JSON e vengono decodificate dal server alla ricezione.
I file non vengono mai cancellati dalla memoria, al fine di essere recuperati all' avvio del server, per non perdere informazioni in seguito ad esempio ad un ipotetico crash del server e quindi per migliorare la robustezza del programma.

### Strutture dati del server e concorrenza

Le strutture dati usate dal server sono in prevalenza delle *HashMap* che mappano stringhe (nomi degli utenti o dei documenti) a variabili o strutture dati di altro tipo, ma sono presenti anche due *FileList*, strutture definite da me in una classe specifica che in pratica sono liste però mantenute anche, sempre aggiornate, su un file di testo il cui nome è stabilito alla creazione della struttura, essendo passato al costruttore di questa. Questa struttura serve per mantenere perennemente in memoria la lista degli utenti registrati e dei documenti creati, in modo da recuperarli in seguito ad un riavvio del server.
Nel dettaglio le strutture sono:

- *usrs*: *FileList* usata per memorizzare in memoria la lista degli utenti, in questo caso scritti come stringhe JSON con solo le voci username e password, ognuna prende una riga del file *users.txt*, che è presente nella cartella dove viene eseguito il programma

- *documents*: *FileList* usata per memorizzare i nomi dei documenti creati sul file *documents.txt* presente nella cartella dove viene eseguito il programma, in questo caso i documenti veri e propri in formato JSON sono memorizzati nella cartella *Documents*, presente anch' essa dentro cartella di esecuzione del programma
- *connUsers*: è una *HashMap* che mappa i nomi degli utenti connessi con le porte usate per comunicare loro gli inviti ricevuti in tempo reale, dato che appunto sono connessi, e viene usata sia per sapere se un utente è connesso, sia per recuperarne la porta su cui comunicare
- *addrs*: è una *HashMap* che mappa i nomi dei documenti presenti in memoria con gli indirizzi usati dai client che collaborano a questi documenti per comunicare in chat, creati al recupero del documento dalla memoria oppure alla creazione dello stesso
- *secmod*: serve per sapere per ogni documento quali sono le sezioni che ne fanno parte che sono in corso di modifica al momento e da quale utente, è forse la struttura più complicata presente, essendo una *HashMap* che mappa i nomi dei documenti in una *HashMap* che a sua volta mappa gli indici delle sequenze ai nomi degli utenti che le stanno modificando
- *notify*: serve per sapere quali notifiche di invito alla collaborazione su un documento bisogna mandare nel momento in cui un utente interessato da qualche invito si connette, è anch' essa una *HashMap* che mappa i nomi degli utenti in array di stringhe contenenti i nomi dei documenti per i quali si è ricevuto un invito
- *doclocks*: questa *HashMap* è usata per la mutua esclusione, mappa infatti i nomi dei documenti con chiavi usate per accederli in mutua esclusione
- *locks*: anche questo *ArrayList*, come si capisce dal nome, è usato per la mutua esclusione, in particolare per mantenere le chiavi da usare per accedere in modo thread-safe a tutte le precedenti strutture dati, ad ogni indice corrisponde la lock per una struttura

Come visto ora, le strutture dati del server sono accedute tutte solo dopo averne acquisito la chiave, in modo tale da non generare situazioni di inconsistenza. Viene usata soltanto una lock per ogni struttura anche se sarebbe stato possibile usare hand-over-hand locking e usarne una per ogni porzione di struttura dati, perchè testato anche con più di mille client il server non si è rivelato particolarmente lento nel gestire le richieste. Le operazioni più complicate e che richiedono più tempo sono quelle di serializzazione e deserializzazione dei documenti ed è quindi preferibile svolgerle il più possibile in contemporanea, quindi non viene usata una singola chiave per regolare l' accesso a tutta la cartella *Documents* ma una per ogni documento e l' associazione tra documenti e chiavi è mantenuta nella struttura *doclocks* descritta in precedenza e il cui accesso per aggiornamenti è a sua volta thread safe.
Il server non termina mai, a meno di chiuderlo forzatamente, ad esempio con Ctrl+c.

## 2. Architettura client e threads utilizzati

Il client usa invece principalmente tre thread, uno, quello con il metodo main, per leggere i comandi ricevuti da linea di comando ed eseguirli comunicando con il server, uno per poter comunicare in chat con gli altri client mentre l' utente sta lavorando al documento insieme ad altri utenti, e infine uno per poter ricevere, mentre è connesso, eventuali inviti per lavorare su altri documenti.
I comandi possono essere più comodamente dati anche tramite un' interfaccia grafica, creata usando librerie di JAVA, in particolare *swing* e *awt*.
Con un' architettura single-thread il client non avrebbe potuto gestire le richieste dell' utente e allo stesso tempo ascoltare i messaggi inviati in chat dagli altri utenti o gli inviti.
Le richieste di invito vengono inviate al server che poi le invia agli utenti interessati.
Gli indirizzi multicast sono generati dal server al momento della creazione di un documento o del recupero di questo dalla memoria e inviati al client in seguito ad una richiesta di *edit*, in seguito il server non è coinvolto nella comunicazione tra client.

Alla fine dell' editing di un documento viene chiuso il thread che gestisce la chat, mentre in seguito ad un logout viene chiuso quello che riceve gli inviti e, se il logout è chiesto mentre è in corso la modifica di un documento, anche quello per la chat.
Il thread listener per gli inviti opera in modalità non blocking e controlla sempre se è stato interrotto, quando lo è chiude il socket con il server, mentre quello per la chat è in ascolto da un gruppo multicast e ogni due secondi controlla se è stato interrotto per eventualmente uscire dal gruppo e chiudere il socket multicast.
Molti controlli per stabilire se un comando, che poi sarà la richiesta al server, è valido, vengono fatti preventivamente dal client, anche se per completezza il server controlla comunque tutto prima di eseguirla.
La modifica del testo di una sezione deve essere effettuata per forza all' interno dell' area specifica nella finestra grafica, anche se poi il comando *end-edit* per fissare la modifica può essere dato da linea di comando. Al contrario l' operazione *receive* può essere richiesta solo tramite linea di comando, perché i messaggi ricevuti vengono mostrati in grafica in tempo reale nel pannello della finestra dedicato alla chat.
Alla creazione della finestra viene anche settato un comportamento da eseguire quando questa viene chiusa, e cioè, se un utente è connesso, inviare una richiesta di *logout* al server prima di terminare il programma.

### Strutture dati del client e concorrenza

Il client utilizza una struttura dati, *notifications*, che ha il solo scopo di memorizzare le finestre di notifica mostrate, in modo tale che se l' utente non le chiude una volta viste, vengano comunque chiuse una volta eseguito un logout o chiuso il programma.
E' presente soltanto un' altra struttura, definita nella classe che implementa il thread listener per la chat, che contiene in messaggi ricevuti dall' utente per mostrarli quando viene chiamata da linea di comando l' operazione *receive* ed è implementata tramite un' *ArrayList*.
Essendo il metodo per la *receive*, che svuota la lista dopo aver mostrato i messaggi, chiamato dal thread principale del client e non dal thread chat listener, che invece li aggiunge una volta ricevuti, è acceduta in mutua esclusione, e l' accesso è gestito tramite *Lock*.
Non è presente una lock che garantisca la mutua esclusione sull' uso del client, quindi le richieste provenienti dalla linea di comando potrebbero sovrapporsi a quelle provenienti dall' interfaccia grafica, ma in un uso normale del client, cioè da un solo utente, questo non può accadere.

## 3. Documenti, utenti e gestione dei file

I documenti sono identificati dal nome e sono formati da sezioni, la stessa sezione non può essere in corso di modifica contemporaneamente da più di un utente.
Inoltre un utente non può richiedere un' operazione *end-edit* per una sezione di un documento se prima non ha chiesto una *edit* sulla stessa.
Gli utenti che non sono stati invitati a collaborare ad un documento dal creatore di questo non possono accedervi per lavorare su una sezione o per vederne il contenuto.
Documenti e utenti sono mantenuti permanentemente in memoria sotto forma di oggetti JSON.
Gli utenti sono memorizzati come stringhe JSON (una per riga) contenenti username e password nel file *users.txt*, mentre per i documenti, la cui rappresentazione JSON è più complessa, è mantenuto solo il nome nel file *documents.txt*, mentre il documento vero e proprio, codificato in JSON, si trova nella cartella *Documents*, creata all' avvio del server oppure recuperata se già esistente.
L' unico vincolo per quanto riguarda gli utenti è che la password deve essere lunga almeno 6 caratteri, che viene controllato dal server, che eventualmente risponde con l' intero che codifica l' errore.
Naturalmente in un contesto più realistico andrebbero aggiunti meccanismi di sicurezza più avanzati come mantenere in memoria le informazioni crittografate usando magari la password come parola chiave e decriptarle una volta recuperate, anche a costo di introdurre overhead al server.

La serializzazione dei documenti in memoria, quando aggiornati o creati, avviene tramite i metodi contenuti nella libreria *nio* di JAVA e lo stesso vale per la deserializzazione, quando questi devono essere recuperati, e anche per l' operazione *show*, in cui il file viene trasferito dal canale del file direttamente su quello del client.
Si usa *nio* anche per recuperare la lista degli utenti o dei nomi dei documenti dal relativo file *.txt*, leggendolo linea per linea e creando una *FileList*.

# 4. Comunicazione Client-Server

Essendo il programma implementato mediante un' architettura client-server, la comunicazione tra queste due entità è fondamentale ai fini del funzionamento dello stesso.
Questa inizia da parte del client che invia una richiesta al server, che la deve gestire.
A parte la richiesta di registrazione di un utente, tutte le richieste sono effettuate tramite normali connessioni TCP.
Questo protocollo garantisce affidabilità per cui non è necessario che né il server né il client implementino meccanismi per garantirla.
La richiesta di registrazione avviene tramite *remote method invocation*, viene esportata l' interfaccia remota che implementa *Remote* in cui è dichiarato solo il metodo *register*, per effettuare la registrazione. In seguito viene creato l' oggetto remoto e registrato in un *registry*, in modo tale che il client possa trovarlo ed invocarne il metodo *register* che fara sì che il thread dedicato si occupi di registrare l' utente, se non ci sono problemi, altrimenti viene restituito un intero che codifica l' errore riscontrato.
Server e client comunicano anche per quanto riguarda le notifiche di invito, cioè quando un utente ne invita un altro a collaborare sul proprio documento, scopo del programma. In questo caso infatti se l' utente è connesso deve ricevere la notifica subito (altrimenti la riceve al momento del login) e per poterlo fare è attivato dal client, al momento della connessione di un utente, un thread che si occupa soltanto di attendere messaggi di invito dal server e mostrarli in una finestra pop-up oppure scriverli sul terminale.
Anche questa comunicazione è effettuata tramite TCP e la porta su cui comunicare è diversa per ogni utente ed è comunicata dal server al momento del login dell' utente. Questo sarebbe superfluo in un contesto realistico poiché con indirizzi IP diversi potrebbe essere usato sempre lo stesso indice di porta, ma usando tutti lo stesso indirizzo, quello locale, è necessario fare così.
Il server opera in modalità non-blocking per poter usare il *selector* e distribuire il carico di lavoro su un pool di thread, ma, una volta partito, il thread worker comunica in modalità blocking con il client, anch' esso in modalità blocking perché hanno bisogno l' uno della risposta dell' altro per procedere.
Il thread usato dal client per ricevere gli inviti invece opera in modalità non-blocking, controllando continuamente di non essere stato interrotto dal client per una richiesta di logout e in quel caso chiudere la connessione con il server e terminare.

## Comunicazione tra client

Gli utenti, mentre collaborano alla modifica di un documento, possono comunicare in chat in multicast per cui il server non è coinvolto, si limita soltanto a generare gli indirizzi di multicast usati, uno per ogni documento, e a comunicarli agli utenti coinvolti al momento di una richiesta di *edit*.
Un utente riceve anche i messaggi che lui stesso invia in modo tale da poter visualizzare o ricostruire, da linea di comando con la *receive*, la storia passata della chat.
Usando UDP non c' è garanzia che tutti i messaggi arrivino a tutti gli utenti, ma in questo caso non è fondamentale, del resto con TCP sarebbero richieste troppe connessioni affidabili che potrebbero rallentare sensibilmente lo scambio di messaggi, che invece deve rimanere veloce e in tempo reale.

# 5. Classi definite

- **FileList**
  Consiste in un' *ArrayList* di stringhe, ma scrive queste stringhe anche su file, una per riga, in modo tale che, in caso di spegnimento o riavvio del server, questa lista possa essere recuperata. Infatti il costruttore prende in input il nome della lista e, prima di creare un file con quel nome, controlla se esiste già e in caso la recupera da questo file. Nel programma viene usata per mantenere la lista dei documenti creati e degli utenti iscritti.
  Contiene i metodi per sapere se un elemento è presente in questa lista e, se lo è, per recuperarlo.

- **User**
  Rappresenta l' utente, una semplice coppia <username, password> e contiene, oltre i metodi per recuperare questi due campi, anche un metodo per ottenere un oggetto JSON che lo rappresenti, in modo che possa essere salvato su file una volta registrato.

- **Document**
  Rappresenta il documento, contiene campi per il nome, per l' utente creatore e quindi amministratore del documento, per gli utenti invitati a collaborarci, per le sezioni e per il numero di sezioni.
  Sono presenti due metodi costruttori, uno per creare il documento partendo solo da username del creatore, nome del documento e numero di sezioni, l' altro per prendere in input un oggetto JSON, decodificarlo ed estrarne tutte le informazioni necessarie a ricreare l' oggetto rappresentante il documento.
  E' presente il metodo per salvarlo in memoria, usato anche per aggiornarlo quando viene modificata una sezione o invitato un utente, che serializza tramite un canale il file in memoria.
  Sono poi presenti i metodi per recuperare il nome, una sezione specifica o il testo dell' intero documento, per aggiungere un utente a quelli a cui è concesso accedervi per vederlo o modificarlo, per aggiornare una sezione, per sapere se un utente specifico è tra quelli invitati, per sapere se contiene una sezione specifica e infine per ottenere l' oggetto JSON che rappresenta il documento stesso.

- **Section**
  Rappresenta una sezione, con i campi testo e indice, infatti le sezioni sono identificate all' interno di un documento dall' indice, partendo dallo zero.
  Dentro il documento, una volta serializzato in memoria, anche le sezioni sono rappresentate come oggetti JSON, per questo è qui presente un metodo per ottenere il JSON che rappresenta la sequenza e il costruttore è anche qui di due tipi, per creare la sezione e per recuperarla da un oggetto JSON.
  Oltre a quello sono presenti i metodi per aggiornarla con un nuovo testo e per recuperare l' indice o il testo della sezione.

- **Command**
  Rappresenta le richieste che l' utente può somministrare al client, contiene i campi per il nome dell' utente che fa la richiesta, per la stringa rappresentante il comando e per le singole parole del comando, collezionate in un' *ArrayList* di stringhe.
  Contiene i metodi per recuperare le singole parole componenti il comando, importanti per ricavare le informazioni utili a elaborare la richiesta, per ottenere la richiesta in formato JSON per inviarla al server e per sapere, tramite i dovuti controlli, se il comando è accettabile o no, perché altrimenti non viene elaborato.

E’ anche presente un metodo per ricavare, in caso di richiesta *send*, tutte le parole dalla terza in poi che in questo caso compongono un messaggio da inviare.

- **GraphicPanel**
  Rappresenta il pannello contenuto dentro alla finestra che rappresenta la parte dedicata alla modifica, visualizzazione, creazione, condivisione dei documenti e visualizzazione della lista di quelli per cui l’ utente ha ricevuto un invito.
  Contiene come componenti le etichette da mostrare in caso di errori vari, le due aree in cui scrivere gli argomenti delle richieste (ad esempio nome del documento e numero della sezione), i pulsanti per inviare i comandi ed infine un’ area di testo grande in cui vengono mostrate la lista dei documenti, il documento per intero o dove può essere modificata la sezione prima di confermare la modifica con un *end-edit*.
  Contiene il metodo per gestire un evento alla pressione di un bottone, creare cioè il comando giusto ed elaborare la richiesta, per mostrare un errore in grafica, per mostrare un testo a richiesta, per prendere il testo dall’ area di modifica e infine per rendere modificabile oppure non modificabile il testo all’ interno dell’ area adibita a tale utilizzo.
  In ogni caso prima di mostrare un errore vengono cancellati i precedenti, perché ogni richiesta non può restituire più di un errore, e quando un’ operazione va a buon fine sono cancellati gli errori precedentemente mostrati.

- **LoginPanel**
  E’ il pannello contenuto nella finestra di login, quella mostrata prima che un utente sia connesso, che consente solo le operazioni di registrazione e di login appunto.
  Alla pressione dei pulsanti viene chiamato il metodo che manda in esecuzione l’ operazione richiesta. Anche qui possono essere mostrati errori, contenuti in etichette, con il metodo apposito e rimossi quando un’ operazione ha successo con l’ altro metodo presente.

- **ChatPanel**
  E’ il pannello, contenuto anch’ esso nella finestra visibile quando un utente è connesso, che contiene l’ area di testo dedicata alla visualizzazione dei messaggi che arrivano in multicast dagli utenti che lavorano insieme a quello connesso ad un documento e l’ area in cui scrivere il messaggio, da cui poi viene anche prelevato, oltre naturalmente al pulsante per inviarlo.
  I metodi servono, oltre che per eseguire l’ azione alla pressione del pulsante per l’ invio del messaggio, per mostrare un testo nell’ area dedicata alla chat (i messaggi della chat, uno per riga), per rimuovere il messaggio dall’ area in cui è scritto una volta prelevato ed inviato e per rimuovere tutti i messaggi ricevuti dall’ area dedicata, quando viene terminata la modifica di un documento.

- **NotificationListener**
  E’ l’ implementazione del thread che si occupa di ascoltare quando arriva un invito dall’ indirizzo locale e su una porta ricevuta dal server al momento del login. Contiene solo il metodo eseguito all’ attivazione del thread, che aspetta un messaggio dal server che conterrà il nome di un documento per il quale si è ricevuto l’ invito, poi crea una finestra di notifica che se non chiusa rimane visibile fino alla chiusura del client o ad un logout.

- **ClientImpl**
  Implementa il client, il metodo *executeAction* esegue la richiesta dell’ utente comunicando con il server, mentre altri metodi servono per creare un oggetto rappresentante un comando da una stringa e per creare le finestre di notifica, login e quella principale quando servono.

Sono presenti anche i metodi per ricevere un intero dal server come risposta, per scrivere e per leggere una determinata quantità di bytes su un buffer con il socket usato dal server.
In questa classe il costruttore non fa niente.

- **Client**
  Contiene il metodo main del client che dopo aver creato un oggetto *ClientImpl* inizia un semplice ciclo in cui può ricevere richieste da linea di comando ed eseguirle.

- **ServerInterface**
  E' l' interfaccia remota usata per la comunicazione RMI per la registrare nuovi utenti.

- **ServerImpl**
  Implementa il server, con tutti i metodi necessari ai thread worker ad eseguire le richieste dei client, ed è qui che è gestita anche la concorrenza.
  Il costruttore qui inizializza tutte le strutture dati necessarie, ed è qui che vengono costruite quelle per documenti e utenti, entrambi recuperati dalla memoria se già presenti i rispettivi file JSON.

- **Task**
  Implementa il thread worker, che viene fatto partire quando il thread listener, tramite un selector identifica un socket sul quale c'è qualcosa da leggere.
  Questo socket viene affidato al thread, che legge la richiesta arrivata dal client e la elabora, continuando la comunicazione con il client fino alla fine di questa.
  Contiene il metodo per leggere e per scrivere una determinata quantità di bytes sul buffer con il client, quello per inviare un messaggio di invito ad un utente online ed un metodo ausiliario per recuperare la prima parola di una stringa

- **Server**
  Implementa il thread listener del server, che crea inizialmente un oggetto *ServerImpl*, esporta un oggetto remoto affinchè possa essere effettuata la registrazione degli utenti tramite RMI, crea la cartella *Documents* se non è già presente e infine inizia il ciclo con il selector in cui accetta connessioni sui socket e affida quelli leggibili ai thread worker.

## 6. Librerie esterne utilizzate

L' unica libreria esterna utilizzata è *Json-simple*, versione 1.1, un package che consente di mappare i tipi principali di JAVA in tipi JSON, consentendomi di usare questo formato per standardizzare utenti, documenti e richieste nel mio programma.

## 7. Istruzioni per l' esecuzione e il testing

Il programma si divide essenzialmente in due parti da compilare: client e server, ed entrambi usano la libreria esterna *json-simple-1.1.jar* e quindi nella compilazione necessitano l' inclusione di questa.
Per il server l' istruzione per compilare è:

*javac -classpath ./json-simple-1.1.jar; Server.java*

mentre quella per mandarlo in esecuzione è:

*java -classpath ./json-simple-1.1.jar; Server*

Per il client invece la compilazione si esegue con:

*javac -classpath ./json-simple-1.1.jar; Client.java*

mentre per l' esecuzione:

*java -classpath ./json-simple-1.1.jar; Client*

Le operazioni possono essere chiamate premendo i pulsanti presenti sull' interfaccia grafica e inserendo nei campi gli argomenti, specificati dalle etichette testuali presenti, quindi il funzionamento usando la finestra è molto semplice da capire una volta aperta.

All' avvio è mostrata la finestra di login per connettersi o registrarsi con i relativi pulsanti e una volta connessi quella principale con tutte le operazioni eseguibili.

La descrizione delle operazioni, insieme alla sintassi dei comandi eseguibili da linea di comando (metodo alternativo per utilizzare il programma), è specificata di seguito:

- **`turing register <username> <password>:`** procedura per registrare un utente al servizio, il server crea un oggetto JSON rappresentante l' utente e lo salva in una riga del file *users.txt*, se c'è un errore lo stampa.
- **`turing login <username> <password>`**: serve per connettersi al servizio, dopo essersi registrati, in questo caso il server aggiunge l' utente a quelli connessi, o restituisce eventuali errori.
- **`turing logout`**: il server disconnette l' utente al momento connesso rimuovendolo dalle strutture dati in cui è presente, il client invece chiude il thread che riceve i messaggi dalla chat e quello che attende gli inviti e torna a mostrare la finestra di login, nascondendo quella principale. Se eseguita mentre è in corso la modifica di una sezione comporta l' annullamento della modifica.
- **`turing create <documento> <numero_sezioni>`**: serve per creare un documento, il server crea un oggetto JSON che rappresenta il documento, a meno che questo non esista già, e lo salva in memoria, aggiungendo anche una riga con il nome del documento nel file *documents.txt*.
- **`turing share <documento> <username>`**: se non ci sono errori invita l' utente specificato a lavorare al documento, può essere eseguita soltanto dal creatore di questo e non dagli altri utenti, nemmeno se precedentemente invitati. Il server riceve la richiesta, aggiorna gli utenti che possono accedere al documento e lo salva in memoria, poi invia l' invito all' utente interessato se online oppure lo salva per mandarlo al momento della connessione di questo se non connesso, infine restituisce l' esito dell' operazione.
- **`turing show <documento>`**: mostra l' intero testo del documento nel terminale oppure nell' area di testo dedicata nella finestra grafica e in fondo comunica anche gli indici delle sezioni del documento che sono attualmente in corso di modifica. Il server qui invia l' intero file JSON, rappresentante il documento, al client, che lo decodifica e ne mostra il contenuto nell' area di testo della finestra, poi invia la lista delle sezioni in modifica, che viene anch' essa mostrata dal client dopo il testo del documento.
- **`turing list`**: serve per vedere a quali documenti l' utente ha il permesso di accedere, il server invia qui una stringa contenente i nomi di questi, con il carattere "a capo" dopo ogni nome in modo da mostrarli uno per riga nell' area di testo della finestra o sul terminale.
- **`turing edit <documento> <indice_sezione>`**: serve a iniziare l' editing di una sezione di un documento. Il server invia il testo di una sezione, il client lo mostra nell' area di testo, resa adesso modificabile, in modo che l' utente lo visualizzi e modifichi. Dopo questa operazione ne può essere eseguita solo una di *end-edit* oppure di *logout*, eseguibile in ogni momento, oltre a quelle per inviare/ricevere i messaggi in chat. Il client infatti in questo caso avvia anche il thread che riceve i messaggi ed eventualmente li mostra nella zona della finestra dedicata alla chat. E' importante ricordare che gli indici delle sezioni partono dallo zero e non dall' uno.
- **`turing end-edit <documento> <indice_sezione>`**: eseguibile solo dopo una *edit*, serve per confermare la modifica di una sezione di un documento. Il client recupera dall' area di testo in finestra, resa a questo punto non più modificabile, il testo presente al momento della richiesta e lo invia al server che aggiorna la sezione corrispondente e salva il documento aggiornato in memoria. Ovviamente questa operazione può essere richiesta solo per una sezione per la quale prima si è richiesta la *edit*, altrimenti si ottiene un errore. Anche il thread listener per la chat viene chiuso dal client e i messaggi della chat rimossi sia dalla grafica sia dalla memoria.

- **`turing send <messaggio>`**: è usata per inviare un messaggio in chat mentre è in corso l' editing di un documento, il testo di questo viene preso dal campo nella finestra se chiamato da qui oppure dai caratteri successivi alla parola "send", se chiamata da terminale. Il server qui non è coinvolto.
- **`turing receive`**: chiamabile solo da terminale, servee per vedere la storia della chat con tutti i messaggi scambiati dagli utenti dal momento dell' ultima *receive* a quello in cui viene evocata. In finestra i messaggi vengono mostrati in tempo reale e sarebbe quindi inutile.
- **`exit/turing exit`**: chiude il client, eseguendo il logout dell' utente che è connesso, se ne è presente uno.

Gli errori eventualmente restituiti dalle qui descritte operazioni vengono tutti, oltre che scritti sul terminale, mostrati sulla finestra, e rimossi quando invece un' operazione ha successo.

## Testing

Nell' archivio contenente il programma è presente anche una classe di test chiamata *ClientTest*.
In questa classe è contenuto un metodo main, in cui vengono generati 32 thread, implementati nella classe *ThreadTest* presente anch' essa nell' archivio, ognuno dei quali implementa un client che fa automaticamente delle richieste in modo da simulare un utilizzo normale del programma.
Ogni thread esegue tutte le operazioni, sbagliando appositamente a volte per mostrare i messaggi di errore sia in grafica sia sul terminale.
Il primo thread ad essere generato, dopo aver creato un documento invita i primi 16 utenti a collaborare al proprio documento per testare il meccanismo di invito, poi prova alcune operazioni che restituiscono errore, gli altri thread invece attendono qualche secondo per dare il tempo al thread zero di portare a termine almeno qualche invito, poi modificano il documento e comunicano in chat mentre lo fanno.
Ovviamente solo i primi sedici, quelli invitati, avranno accesso al documento, gli altri riceveranno accesso negato, inoltre questi sedici dovrebbero ricevere un invito prima di avere il tempo di connettersi per poi mostrarlo al momento del login, ma questo non può essere garantito a prescindere e quindi potrebbero riceverlo dopo, mentre sono online.
Infine questi thread richiedono di visualizzare il documento, cosa concessa solo a chi avrà ricevuto l' invito, e richiederanno la lista dei documenti a cui hanno accesso.
Questo programma di test terminerà solo quando tutti i thread avranno terminato la loro esecuzione (con un logout), a quel punto, prima di terminare, procederà all' eliminazione di tutti i file residui lasciati dal server, visto che appunto è solo un test e non è necessario che i documenti creati qui siano mantenuti.
Il test si compila con:

*javac -classpath ./json-simple-1.1.jar; ClientTest.java*

e si esegue con:

*java -classpath ./json-simple-1.1.jar; ClientTest*