Turing (disTribUted collaboRative edItiNG)
Progetto Reti 2018/19

Alexandra Bradan [530887]

12 novembre 2019

Indice

1 Panoramica delle classi definite

1.1 Server

1. *TuringServer*: è la classe da mettere in esecuzione per avviare il Server;
2. *ServerConfigurationsManagement*: classe che si occupa di fare il parsing del file di configurazione del Server e memorizzare i valori estrapolati da esso;
3. *MyExecutor*: classe che estende la classe *ThreadPoolExecutor*, per personalizzare il ThreadPool che gestisce le richieste dei Clients ed assegna ai suoi worker-threads dei nomi significativi;
4. *TuringListener*: classe che implementa l'interfaccia *Runnable* (task messo in esecuzione dal *TuringServer*) e designata, tramite un ServerSocket ed un Selector, a monitorare e gestire le connessioni e le richieste dei Clients;
5. *TuringWorker*: classe che implementa l'interfaccia *Runnable* e che rappresenta un task che un worker-thread del ThreadPool dovrà soddisfare;
6. *TuringTask*: classe che raccoglie i metodi per soddisfare le operazioni che il servizio fornisce;
7. *ServerMessageManagement*: classe che si occupa di implementare la logica di lettura delle richieste provenienti dai Clients e quella dell'invio delle risposte;
8. *ServerDataStructures*: classe che raccoglie tutte le strutture dati utilizzate dal Server;
9. *TuringRegistrationRMIInterface*: interfaccia che estende la classe *Remote* per poter esportare il metodo remoto tramite il quale gli utenti possono registrarsi al servizio;
10. *TuringRegistrationRMI*: classe che implementa l'interfaccia *TuringRegistrationRMIInterface* e fornisce la logica di registrazione alla piattaforma;
11. *User*: classe che raccoglie le informazioni caratterizzanti di un utente registrato al servizio;
12. *Document*: classe che raccoglie le informazioni caratterizanti di un documento creato e memorizzato sulla piattaforma;
13. *MulticastAddressRandomGenerator*: classe che permette di generare indirizzi di multicast, da assegnare ai documenti creati e consentire di aprire chat per far comunicare gli utenti che editano sezioni differenti di uno stesso documento;
14. *ServerShutdownHok*: classe che implementa l'interfaccia *Runnable* e che implementa la logica di terminazione del Server, eseguendo un GRACEFUL SHUTDOWN.

1.2 Client

1. *TuringClient*: è la classe da mettere in esecuzione per attivare un Client;
2. *ClientConfigurationsManagement*: classe che si occupa di fare il parsing del file di configurazione del Client e memorizzare i valori estrappolati da esso;
3. *ClientCommandLineManagement*: classe che si occupa di leggere i comandi da linea di tastiera, attraverso l'utilizzio di un *BufferedReader*;
4. *ClientMessageManagement*: classe che si occupa di implementare la logica di invio delle richieste al Server e della lettura ed interpretazione delle risposte provenienti dal Server. A seconda della risposta pervenuta, tale classe si occupa di effettuarne una stampa personalizzata, indicando se un utente è connesso, chi è connesso in quel dato momento;
5. *ClientInvitesListener*: classe che estende la classe *Thread* e designata a rimanere in ascolto degli inviti di collaborazione a documenti;
6. *ClientChatListener*: classe che estende la classe *Thread* e designata a rimanere in ascolto dei messaggi della chat associata al documento che un utente, eventualemente, sta editando;
7. *ClientShutdownHok*: classe che implementa l'interfaccia *Runnable* e che implementa la logica di terminazione del Client, eseguendo un GRACEFUL SHUTDOWN.

1.3 Classi comuni

1. *FunctionOutcome*: enum che raccogli il possibile esito dell'esecuzione di una funzione: successo oppure fallimento. Tale risultato dipende dalla semantica e logica implementativa delle singole funzioni;
2. *FileManagement*: classe che raccoglie tutti i metodi utili per gestire files e directories, utilizzando *Java NIO 2 File API*;
3. *ServerResponse*: enum che raccoglie tutte le possibili risposte che il Server può fornire ai Clients;
4. *CommandType*: enum che raccoglie tutte le possibili richieste che un Client può sottomettere al Server;
5. *SocketChannelReadManagement*: classe che implementa la logica di lettura da un SocketChannel;
6. *SocketChannelWriteManagement*: classe che implementa la logica di scrittura su un SocketChannel.

2 Formato dei messaggi

Client e Server si scambiano messaggi che utilizzano un formato pressoché identico. Il messaggio è così costituito:

- HEADER da 8 bytes:
  - i primi 4 bytes codificano un intero che corrisponde al tipo di richiesta, nel caso del Client (tutte le richieste sono espresse sottoforma di *enum* e si trovano nella classe *CommandType*) e al tipo di risposta, nel caso del Server (tutte le risposte sono espresse sottoforma di *enum* e si trovano nella classe *ServerResponse*);
  - i restanti 4 bytes codificano la dimensione del PAYLOAD del messaggio. Se l'HEADER è fine a se stesso ed è utilizzato solamente come messaggio di acknowledgment, il PAYLOAD sarà vuoto, di conseguenza la sua dimensione sarà pari a zero e verrà notificato per consentire all'altra entità di non attendere il suo invio, altrimenti la notifica della dimensione del corpo del messaggio farà preparare l'altra entità alla sua ricezione (allocazione di un ByteBuffer di dimensione pari a quella comunicata)).
- PAYLOAD di dimensione variabile. A seconda che si tratti di passare al Server gli argomenti per una richiesta, oppure di spedire il contenuto di un file dal/per il Server, tale campo racchiude sempre un'istanza della classe *String*, di cui l'HEADER ne raccoglie la dimensione in bytes (i canali di comunicazione lavorano con stream di bytes).

3 Struttura del progetto

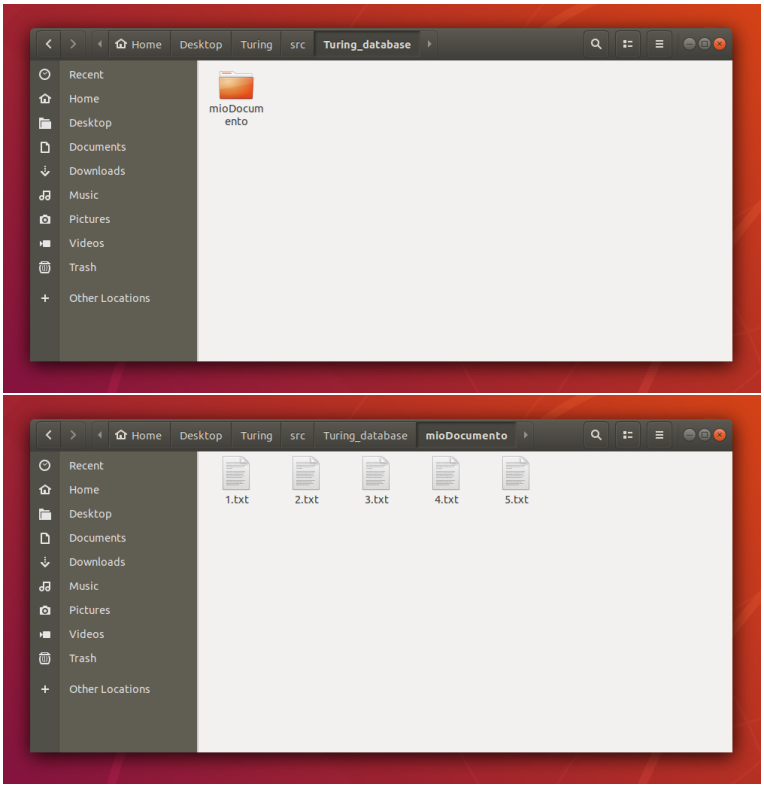
3.1 Server

Come in qualsiasi paradigma client/server, il primo processo da mettere in esecuzione è il TuringServer, il quale si occupa di:

1. leggere e fare il parsing del file di configurazione per attivare il servizio. Tale file è reperibile nella cartella "/data/turingServer.conf" del progetto. In alternativa è possibile specificare il path assoluto di un proprio file di configurazione, passandolo come argomento quando si mette in esecuzione il Server. Il formato del file di configurazione deve rispettare quello di default. La non conformità a tale formato rende il parsing delle variabili di configurazione impraticabile e di conseguenza alla terminazione del processo;

2. se il parsing del file di configurazione ha avuto successo e tutte le variabili di configurazione necessarie sono state istanziate, viene creata la cartella "/Turing_database/" se non esiste (se esistente la cartella è stata svuotata al termine della precedente esecuzione del Server. Il servizio non implementa la persistenza dei dati, in quanto non menzionato dalla specifica). Tale cartella viene utilizzata dal Server per memorizzare i documenti (sottoforma di cartelle) e le relative sezioni (sottoforma di file) degli utenti:

- i documenti qui presenti hanno i nomi loro assegnati dal creatore, a patto che non contengano caratteri speciali non consentiti per la creazione di directories. Se il nome contiene uno o più caratteri speciali viene notificato un errore al creatore e gli si chiede di scegliere un nome privo di essi;
- le sezioni di ogni documento prendono il nome dalla numerazione corrispondente alla sua creazione (quindi la sezione i di un documento sarà un file denominato *i.txt*);



Esempio di come viene memorizzato il documento *mioDocumento*, formato da cinque sezioni.

3. allocare le strutture dati utilizzate dal servizio (vedere la sezione *Strutture dati e gestione concorrenza* per maggiori informazioni al riguardo);

4. creare ed attivare un FixedThreadPool, il quale utilizza un numero finito di worker-threads per leggere, soddisfare e rispondere ai Clients che sottomettono richieste. Si è optato per utilizzare un FixedThreadPool (il numero di workers da attivare viene estrapolato dal file di configurazione) per evitare che troppe richieste potessero portare ad una sovrattivazione di worker-threads ed al conseguente ribasso del QoS (*Quality of Service*) per i Clients, causato dalle diminuzioni delle prestazioni di ogni worker (scenario manifestabile nel caso di un CachedThredPool). La creazione del ThreadPool viene, inoltre, personalizzata estendendo la classe *ThreadPoolExecutor* (la classe che fa questo è *MyExecutor*), per consentire di dare un nome specifico ai threads al suo interno (ogni worker del Th-

readPool si chiama *Worker_i*, dove *i* corrisponde alla posizione occupata nel ThreadPool). Questo consente di effettuare stampe personalizzate su quale thread stia soddisfacendo la richiesta di un particolare Client;

5. creare ed attivare il thread TuringListener, che si occuperà della connessione dei Clients al servizio e del soddisfacimento delle loro richieste;

6. creare ed attivare il thread ServerShutdownHook, passandogli come riferimento il ThreadPool, il thread TuringListener e la classe ServerConfigurationManagement, che memorizza le variabili di configurazione del Server. Tale thread, infatti, ha il compito di implementare la logica di terminazione del Server, effettuando un GRACEFUL SHUTDOWN (per maggiorni informazioni sulla terminazione del Server e su come agisce lo ServerShutdownHook nello specifico, consultare la sezione *Gestione terminazione*).

Creato ed attivato il thread TuringListener, la scena passa a quest'ultimo, che provederrà:

1. a creare lo stub RMI per consetire la registrazione, da remoto, degli utenti alla piattaforma;

2. ad aprire il ServerSocketChannel utilizzato per ascoltare il sopraggiungere di connessioni da parte dei Clients (il ServerSocketChannel è settato in modalità non-blocking per poter essere registrato al selettore);

3. all'apertura di un Selector per poter iterare sui canali pronti per un'operazione di I/O;

4. alla regitrazione del ServerSocketChannel al Selector per poter monitorare tramite di esso i SocketChannels dei Clients che chiedono di connettersi al servizio;

5. all'attivazione del ciclo di ascolto (idealmente eterno, dato che si presuppone che il servizio rimanga sempre attivo, ma terminabile in concomitanza del sopraggiungere dei segnali SIGINT, SIGQUIT o SIGTERM), nel quale tramite il Selector:

   (a) si monitorano i SocketChannels dei Clients che richiedono di connettersi per la prima volta al servizio. Il ServerSocketChannel li registra al Selector per monitorare il sopraggiungere di richieste future(i SocketChannel accettati vengono settati in modalità non-blocking per poter essere registrati al selettore e monitorati tramite di esso);

   (b) si monitorano le richieste che sopraggiungono sui SocketChannels precedentemente accettati e registrati dal ServerSocketChannel al selettore. Quando viene rilevata la richiesta da parte di un Client, il relativo SocketChannel viene rimosso temporaneamente dall'insieme dei canali monitorati dal selettore, per darlo in affidamento ad un worker del ThreadPool (se libero, altrimenti il suo SocketChannel viene messo all'interno della WorkingQueue, dalla quale verrà recuperato quando un worker avrà terminato di soddisfare qualche richiesta). Il worker che prenderà in affidamento il SocketChannel dovrà:

i. leggere la richiesta dal SocketChannel;
ii. soddisfare la richiesta del Client invocando un metodo opportuno (tutti i metodi che soddisfano le operazioni fornite dal servizio si trovano nella classe *TuringTask*);
iii. mandare una o più risposte di esito (il quantitativo dipende dalla richiesta effettuata) dal Client;
iv. inserire il SocketChannel nella coda dei canali da re-inserire nel selettore (*selectorKeysToReinsert* presente nella classe *ServerDataStructures*). Tale coda è una *BlockingQueue* perchè utilizzata da tutti gli workers e serve per implementare il rinserimento dei SocketChannel che hanno ricevuto una risposta di esito alla loro richiesta e di conseguenza, possono sottometterne un'altra. Tale coda viene consultata periodicamente dal *TuringListener*, allo scadere del timer impostato sulla *select* del Selector (l'operazione della selezione dei canali pronti non è bloccante, ma viene settato un timeout per verificare, periodicamente, la presenza di canali da ri-registrare al selettore. Ricordiamo infatti, che una richiesta porta alla rimozione temporanea del canale dal selettore e quando la richiesta è stata soddisfatta e il suo esito inviato, è necessario rimettere il canale nel Selettore per poter ascoltare altre sue richieste).

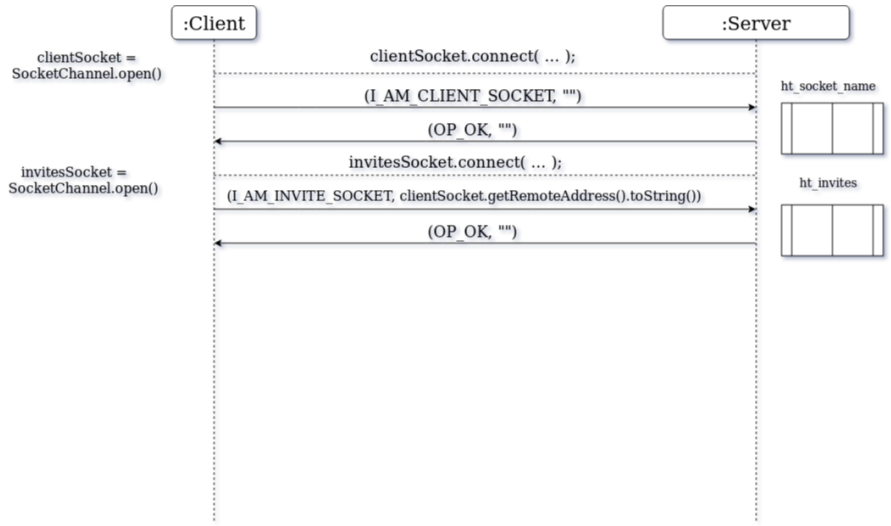
3.2 Client

Quando un TuringClient entra in esecuzione si occupa di:

1. leggere e fare il parsing del file di configurazione per attivare il servizio. Tale file è reperibile nella cartella "/data/turingClient.conf" del progetto. In alternativa è possibile specificare il path assoluto di un proprio file di configurazione, passandolo come argomento quando si mette in esecuzione il Client. Il formato del file di configurazione deve rispettare quello di default. La non conformità a tale formato rende il parsing delle variabili di configurazione impraticabile e di conseguenza alla terminazione del processo. Si è preferito differenziare il file di configurazione del Client da quello del Server, per poter personalizzare e specializzare il comportamento del Client.

2. aprire un SocketChannel e connetterlo al Server per poter comunicare con esso, sottomettendo richieste ed attendendo risposte di esito;

3. aprire un SocketChannel e connetterlo al Server, specificandogli che questo SocketChannel è da utilizzarsi come canale di invio inviti del primo SocketChannel connesso. In questo modo il Server si crea l'associazione tra il primo SocketChannel (utilizzato per leggere richieste ed inviare risposte) e il secondo SocketChannel (utilizzato per spedire gli inviti alle collaborazioni, relativi all'utente connesso con il primo SocketChannel). Il protocollo utilizzato per fare questo è riassunto nella seguente figura:

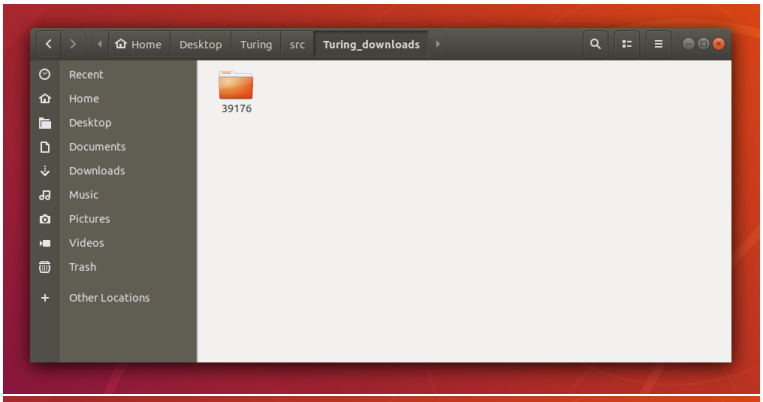


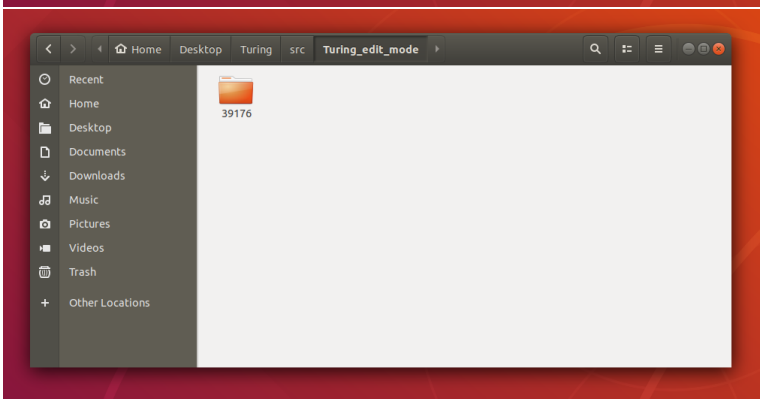
La figura mostra come il Server tiene traccia dell'associazione tra i due SocketChannels. Il primo SocketChannel a connettersi al Server è il *clientSocket*, utilizzato per inviare e leggere messaggi. Se la connessione ha avuto successo, il *clienSocket* manda un primo messaggio al Server notificandogli che lui è il SocketChannel principale. Il Server tramite il metodo *getRemoteAddress().toString()* reperisce l'indirizzo e la porta di questo SocketChannel ed inserisce la coppia (indirizzo_remoto, clientSocketChannel) nella *hashtable_socket_names*. Il Server manda un messaggio di acknowledgment al Client, che di conseguenza provvede a connettere il secondo SocketChannel, ossia l' *invitesSocket*, utilizzato per leggere le notifiche d'invito alle collaborazioni. L' *invitesSocket* notifica al Server che lui, invece, è un SocketChannel di invio inviti e specifica nel body del suo messaggio a quale SocketChannel si riferisce. Quando il Server legge il suo messaggio, dal body estrappola l'indirizzo remoto del Socket a cui si rapporta, utilizza questo indirizzo come chiave per reperire il clientSocket relativo dalla *hashtable_socket_names* ed inserisce la coppia (clientSocket, invitesSocket) nella *hashtable_invites*. A questo punto ha creato l'associazione tra primo SocketChannel e secondo SocketChannel, può rimuovere la entry (indirizzo_remoto, clientSocketChannel) dalla prima HashTable e da ora in avanti quando un worker dovrà notificare un invito al Client, reperirà il canale sul quale sottomettere questo invito dalla seconda HashTable. L'associazione tra i due SocketChannels rimarrà per tutto il ciclo di vita dei due Socket e non appena uno dei due termina il Server provvede a rimuoverla.

Si è scelto di implementare il canale di invio inviti come un normale SocketChannel (invece che utilizzare un ServerSocketChannel ed un Selector per monitorare canali multipli), perchè l'invio di inviti alla collaborazione risulta essere un'operazione sporadica e facilmente gestibile con un solo canale di comunicazione (si necessita solamente di un *synchronized statement* sul canale per gestire la scrittura, in mutua esclusione, su di esso da parte di workers diversi). Ad attendere il sopraggiungere degli inviti su questo SocketChannel viene attivato il thread ClientInvitesListener, che

in un ciclo di attesa, legge uno ad uno gli inviti inviati e gli stampa prontamente, notificandoli al Client. Se un utente non dovesse essere online al momento dell'invito, il worker inserirà, molto semplicemente, l'invio all'interno dell'insieme degli inviti pendenti a lui associato e quando l'utente farà nuovamente il LOGIN, gli inviti pendenti gli veranno notificati come descritto sopra;

4. se il parsing del file di configurazione e la creazione dei due SocketChannels hanno avuto successo, vengono create le cartelle "/Turing_downloads/" e "/Turing_edit _mode/" se non esistenti. Ogni Client popola queste due cartelle con una cartella a lui dedicata. Quindi per ogni Client abbiamo una cartella a lui dedicata nella directory "/Turing_downloads/" (dove il Client andrà a memorizzare i files scaricati a seguito delle operazioni SHOW_DOC e SHOW_SEC) ed una cartella a lui riservata nella directory "/Turing_edit _mode/" (dove il Client andrà a memorizzare le sezioni da editare a seguito dell'operazione EDIT). Tali cartelle vengono battezzate con la porta con la quale il primo SocketChannel comunica con il Server, per avere una nomenclatura univoca. Quindi, per esempio, se abbiamo un Client che si è connesso al Server con un SocketChannel che rimane in ascolto sulla porta 39176, le due cartelle avranno il seguente contenuto:





Tali cartelle vengono utilizzate dal Client per tutto il suo ciclo di vita (anche se si connettono diversi utenti con il medesimo Client) e vengono cancellate al momento della sua terminazione (questo verrà fatto dal ClientShutdownHook);

5. istanziare un riferimento al thread ClientChatListener. Tale thread servirà per inviare e ricevere i messaggi sulla chat associata al documento editato dall' utente. Verrà quindi effettivamente attivato da un commando di EDIT e disattivato da un comando di END-EDIT, ma il riferimento serve per passarlo al thread ClientShutdownHook e poter fare un GRACEFUL SHUTDOWN del Client in caso di anomalia / uscita volontaria (END-EDIT). Come da specifica, l'invio e la ricezione di messaggi sulla chat associata ad un documento avviene utilizzando il protocollo UDP. A tal proposito il thread ClientChatListener, alla sua attivazione, crea un MulticastSocket (il *chatSocket*), con il quela si registra all'indirizzo ed alla porta di multicast (*multicast group*) associati al documento, per poter ricevere i messaggi inviati sulla chat di quest'ultimo. Siccome l'ascolto dei messaggi sulla chat prevede che il ClientChatListener attivi un ciclo di ascolto dei medesimi e siccome l'operazione di *receive* con la quale il ClientChatListener si mette in ascolto sul *chatSocket* è bloccante, per poter uscire dal ciclo di ascolto quando l'utente termina di editare il documento oppure sopraggiunge qualche anomalia e ci si deve apprestare alla terminazione, viene impostato un timer sul DatagramSocket per risvegliarlo periodicamente e controllare se uno dei due scenari soprastanti si è manifestato ed eventulmente terminare il ciclo di ascolto, oppure continuarlo fino al prossimo timeout. L'invio dei messaggi sulla chat, invece, passa tramite il Server:

   - sia per far rimanere le funzionalità che il Client esegue, minimali;
   - sia per aggiungere il concetto, mancante nel protocollo UDP, di *safeness* (in questo modo viene assicurato al Client l'invio e la ricezione dei messaggi);
   - sia per proteggere il MulticastSocket da più scritture simultanee su di esso (lato Server è stato possibile fare questo, mediante l'utilizzo di un *synchronized statement* su di esso);

   Quindi, nel caso in cui l'utente voglia inviare un messaggio sulla chat, dovrà sottomettere tale richiesta ed il contenuto del messaggio che desidera inviare, al Server. È, infatti, per mezzo di un worker del ThreadPool che avviene l'invio di un messaggio sulla chat: il worker crea un DatagramChannel e senza bisogno di registrarsi (l'invio di messaggi sul DatagramSocket non richiede la registrazione al *multicast group*), può inviare il messaggio in multicast.

6. creare ed attivare il thread ClientShutdownHook, passandogli come riferimento il SocketChannel utilizzato per inviare richieste e leggere risposte, il thread ClientInvitesListener ed il thread ClientChatListener. Tale thread, infatti, ha il compito di implementare la logica di terminazione del Client, effettuando un GRACEFUL SHUTDOWN (per maggiori informazioni sulla terminazione del Client e su come agisce lo ClientShutdownHook nello specifico, consultare la sezione *Gestione terminazione*).

7. attivare il ciclo di lettura dei comandi da tastiera e fare delle validazioni sintattiche sui comandi inseriti (presenza di tutti gli argomenti necessari, verifica, dove necessario, che il numero di sezione/i sia un valore numericamente interpretabile). Se un comando inserito risulta valido, il Client lo

invia (incapsulandolo in un messaggio opportuno) al Server ed a seconda delle casistiche, attende una o più risposte di esito (il clientSocket è settato in modalità blocking, ecco perchè c'è una temporizzazione tra invio di richieste ed attesa di risposte).

4 Strutture dati e gestione della concorrenza

4.1 Server

Lato Server, le strutture dati utilizzate per implementare il servizio (presenti nella classe *ServerDataStructure*) hanno le seguenti segnature:

1. ConcurrentHashMap<SocketChannel, String>online_users: tabella hash che raccoglie gli utenti connessi, in un dato momento. Ogni entry della hashtable contiene l'associazione tra il SocketChannel e l'username di un utente connesso;

2. ConcurrentHashMap<String, User>hash_users: tabella hash che raccoglie gli utenti registrati alla piattaforma. Ogni enty della hashtable contiene l'associazione tra l'username di un utente registrato e l'istanza della classe *User* che lo caratterizza, ossia:

   - username;
   - password;
   - insieme dei documenti che è abilitato a modificare, in quanto creatore/collaboratore;
   - insieme degli inviti pendenti (arrivati quando l'utente era offline);
   - eventuale sezione del documento che sta editando.

3. ConcurrentHashMap<String, Document>hash_document: tabella hash che raccoglie i documenti memorizzati sulla piattaforma.Ogni enty della hashtable contiene l'associazione tra il nome del documento e l'istanza della classe *Document* che lo caratterizza, ossia:

   - nome del documento;
   - username del creatore del documento;
   - numero di sezioni che possiede;
   - indirizzo di multicast ad esso associato;
   - insieme degli utenti che hanno i permessi per editarlo;
   - array di stringhe che permette l'editing in mutua esclusione delle sezioni del documento. Ogni slot di tale array contiene un riferimento libero se la sezione non è editata da nessuno, altrimenti contiene l'username dell'utente che la sta editando. L'acquisizione e il rilascio della mutua esclusione vengono fatti tramite due metodi *synchronized*;

4. ConcurrentHashMap<String>hash_multicast: tabella hash utilizzata per memorizzare gli indirizzi di multicast generati ed assegnati ai documenti creati, per poter aprire una chat e tramite il protocollo UDP, dare vita

ad una comunicazione tra tutti gli utenti che stanno editando una sezione diversa dello stesso documento. La generazione dell'indirizzo di multicast è affidata alla classe *MulticastAddressRandomGenerator*. Tale indirizzo viene assegnato nel momento della creazione di un documento e rimane associato al documento per tutto il suo periodo di vita. L'assegnazione dell'indirizzo è statica, perchè si è reputato sufficientemente ampio il range di indirizzi di classe D (multicast) assegnabili:

$$2^{28} - 256$$

ossia gli indirizzi che vanno dal range [224.0.1.0, 239.255.255.255], con l'esclusione del range di indirizzi [224.0.0.0, 224.0.0.255] che vengono utilizzati per il RIPv2 (Routing Information Protocol);

5. ConcurrentHashMap<String, SocketChannel>hash_socket_names: tabella che contiene l'associazione tra l'indirizzo remoto di un *clientSocket* ed il SocketChannel stesso. Come descritto nell'implementazione soprastante dell'invio inviti ad un utente online, tale hashtable è da utilizzare come "trampolino" per costruire, l'utilizzata, *hash_invites*;

6. ConcurrentHashMap<SocketChannel, SocketChannel>hash_invites: tabella hash utilizzata per memorizzare l'associazione tra un SocketChannel ed il relativo SocketChannel da utilizzare per inviare inviti;

7. BlockingQueue<SocketChannel>selectorKeysToReinsert: coda che contiene i SocketChannels che sono stati rimossi, temporaneamente, dal selettore per leggere, soddisfare e ricevere una risposta di esito ed ora necessitano di esservi re-inseriti per poter sottomettere nuove richieste.

8. BlockingQueue<SocketChannel>selectorKeysToDelete: coda che contiene i SocketChannels da eliminare dal selettore, perchè si tratta di canali usati solo per inviare inviti (si tratta di *idle channels*, ossia canali inattivi, connessi al Server solo per notificargli la loro esistenza, ma che non invieranno mai richieste (vengono utilizzati solo dagli workers per spedire inviti agli *clientSockets* relativi)).

Tutte le strutture dati sono delle *Concurrent Collections* appartenenti alla classe *java.util.concurrent*, che implementano meccanismi di lock implicite (come nel caso delle *BlockingQueue*) oppure di *lock striping* (come nel caso delle *ConcurrentHashMap*). In particolare:

- le *BlockingQueue* garantiscono l'accesso in mutua esclusione all'intera coda, quando si vuole inserire oppure rimuovere un elemento;

- le *ConcurrentHashTable* garantiscono l'accesso ai più buckets in mutua esclusione, andando ad inserire una nuova coppia *(chiave, valore)* solo se la chiave è assente e la rimozione di una coppia *(chiave, valore)* solo se la chiave è presente. Questo garantisce l'univocacità degli usernames, dei nomi dei documenti e degli indirizzi di multicast utilizzati dalla piattaforma.

Aspetti importanti sulla concorrenza della piattaforma, inoltre, sono:

1. la scrittura, in mutua esclusione, da parte di più workers di un invito sull' *invitesSocket* dello stesso Client. L'accesso al canale viene protetto da un *synchronized statement* su di esso, per assicurare che uno ed un solo worker invii un invito su di esso, in un determinato istante di tempo;

2. come è stato ribadito nella sezione *Struttura del progetto*, l'invio dei messaggi sulla chat associata ad un documento passa tramite il Server, per mediare la casistica in cui più Clients cerchino di scrivere sullo stesso DatagramSocket contemporaneamente. Delegando la scrittura al Server, si può proteggere l'accesso al canale con un *synchronized statement* su di esso, per assicurare che uno ed un solo worker invii un messaggio su di esso, in un determinato istante di tempo.

3. la cartella "/Turing_database/" viene acceduta da tutti gli workers del ThreadPool. Di conseguenza tutte le letture e le scritture sulle varie sezioni/-files vengono fatte tramite l'utilizzo di *FileLock* appartenenti al package *java.nio.channels.FileLock* ed utilizzabili per mezzo di un *FileChannel*.

4.2 Client

Lato Client, l'unica struttura dati necessaria è una coda per memorizzare i messaggi che sopraggiungono sulla chat di un documento che si sta editando. La lettura dei messaggi da parte dell'utente è infatti asincrona, ossia avviene tramite un'esplicita richiesta da parte dell'utente, come commissionato dalla specifica nel caso di un'interazione con il Server a riga di comando, come l'implementazione da me redatta . A tal proposito, il ClientChatListener inserisce i messaggi che gli arrivano all'interno di una BlockingQueue che funge da history (da qui il nome *history* per designarla). Tale coda necessita di mutua esclusione, perchè bisogna mediare l'inserimento dei messaggi da parte del ClientChatListener e la richiesta di lettura e conseguente rimozione di messaggi, da parte del TuringClient.

Come abbiamo visto, ogni Client ha una cartella dedicata sia nella directory "/Turing_downloads/" che in "Turing_edit_mode" e di conseguenza non sono necessarie locks per accedere ai files (ogni Client accede esclusivamente ai rispettivi).

5 Gestione della terminazione

Sia il Server che il Client utilizzano il meccanismo dello ShutdownHook. Uno Shutdown Hooks è un costruttore speciale, che permette di collegare un pezzo di codice da eseguire quando la JVM è in terminazione. Questo meccanismo permette di specificare, quindi, delle operazioni di clean up quando un programma termina in modo anomalo oppure volontariamente invocando *System.exit()*. Per implementare questo meccanismo bisogna creare una classe che estente la classe *java.lang.Thread* ed inserire la logica che si vuole operare quando la VM è in chiusura all'interno del metodo *run()*. Dopodiché si registra un'istanza di questa classe come ShutdownHook della JVM, chiamando il metodo *Runtime.getRuntime().addShutdownHook(Thread)*. Il Client e il Server fanno

questo, rispettivamente nella classe *ServerShutdownHook* e in *ClientShutdownHook* e tutto questo contribuisce ad un GRACEFUL SHUTDOWN da parte di entrambe le entità.

5.1 Server

Al *ServerShutdownHook* vengono passati i seguenti riferimenti:

1. il thread TuringListener che contiene il ServerSocket e il Selector (siccome sono aperti in un *try-with-resources* e quindi sono dichiarati come risorse del programma, vengono chiusi dal *try* stesso, quando il TuringListener termina);
2. il ThreadPool, con gli workers attivi.
3. la classe ServerConfigurationManagement, che contiene le variabili di configurazione estrapolate dal file di configurazione del Server.

Lo *ServerShutdownHook* provvede ad invocare il metodo *shutdown()* per far terminare il ThreadPool (il metodo ha un comportamento blando, perchè il ThreadPool non viene terminato immediatamente, ma si provvede a rifiutare nuove richieste, a soddisfare le richieste pendenti nella coda di lavoro ed una volta esaurite, all'attesa della terminazione di tutti gli workers). Successivamente viene svuotata la cartella "/Tuirng_database/", che memorizzava i documenti creati dagli utenti (non si implementa la persistenza dei dati, perchè non menzionato nella specifica). Infine viene fatta la *join* sul TuringListener.

5.2 Client

Al *ClientShutdownHook* vengono passati i seguenti riferimenti:

1. il *clientSocket* collegato al Server per inviare richieste e leggere risposte;
2. il thread ClientInvitesListener, contenente il riferimento all' *invitesSocket* su cui ascolta il sopraggiungere di inviti di collaborazione;
3. il thread ClientChatListener, contentente il riferimento al *chatSocket* su cui ascolta il sopraggiungere di messaggi sulla chat relativa al documento che eventualmente l'utente sta editando (in caso contratio, il thread non è attivo);
4. la classe ClientConfigurationManagement che contiene le variabili di configurazione estrapolate dal file di configurazione del Client.

Lo *ClientShutdownHook* provvede a chiudere il *clientSocket*, a far uscire il ClientInvitesListener dal ciclo di ascolto e fargli chiudere l' *invitesSocket*, a far uscire il ClientChatListener dal ciclo di ascolto (se eventualmente era attivo) e fargli chiudere il *chatSocket* ed a cancellare le cartelle dedicate al Client, presenti nelle directories "/Turing_downloads/" e "/Turing_editing_mode/".

Lato Server, quando un Client termina, il Server se ne accorge dall'impossibiltà di leggere una nuova richiesta su di esso. Si predispone quindi a:

- rimuoverlo dalla hasttable degli utenti online (se l'utente che lo utilizzava era connesso);

- liberare l'eventuale documento editato dall'utente;
- eliminare l'associazione tra il *clientSockte* e il suo *invitesSocket* dalla *hash_invites* (qui notiamo come il SocketChannel di invio inviti sia legato al SocketChannel utilizzato per inviare richieste e leggere risposte, per tutto l'arco della sua vita). Se è l'*invitesSocket* a fallire, viene fatto terminare anche il Client e la chiusura del *clientSocket* ci riporta nello scenario appena descritto.

6 Testing

Per sviluppare e testare il progetto è stato utilizzato l'ambiente di sviluppo integrato *IntelliJ IDEA 2019.2.3*. Il progetto è stato, altresì, testato ed avviato a linea di comando sulle seguenti piattaforme:

1. Ubuntu 18.04.2 LTS
2. Microsoft Windows 10 Home, versione a 64-bit

Di seguito verranno brevemente illustrate le interazioni tra il Client ed il Server ed i tests effettuati.

Un' assunzione da compiere è quella che i comandi vengano validati sintatticamente dalla classe *ClientCommandLineManagement*, che si occupa di verificare che le operazioni digitate siano pertinenti al servizio offerto ed abbiano il corretto numero e tipo di argomenti. Di conseguenza, quando un Client sottomette una richiesta, essa è sintatticamente corretta.

6.1 Register

La registrazione di un utente avviene tramite la chiamata del metodo remoto messo a disposizione dal Server e consiste nell'inserire all'interno della *hash_users*, la entry (username, istanza di *User*), se non esiste già un bucket con la medesima chiave e se il Client che si sta utilizzando non è già connesso al servizio con un altro username (bisogna prima fare il logout, per poter registrare un nuovo utente).

Test effettuati:

1. sign in di un utente già registrato;
2. sign in di un utente, mentre un'altro utente è connesso, utilizzando lo stesso Client;
3. sign in di un utente non registrato.

6.2 Login

La connessione di un utente avviene tramite la verifica che l'utente non sia già connesso, per poi passare a verificare che sia registrato alla piattaforma, successivamente si verifica se la password fornita corrisponde a quella contenuta nella sua istanza *User* ed infine si effettua l'inserimento all'interno della *hash_onlines* dell' entry ((clientSocket), username), se non esiste già un bucket con il medesimo valore (utente già connessosi, nel frattempo, con un altro

Client). Inoltre, il Server controlla se ci sono inviti pendenti e in caso affermativo, reperisce dalla *hash_invites* l'associazione tra il *clientSocket* ed il relativo *invitesSocket* (se l'utente è arrivato a sottomettere la richiesta, tale associazione esisterà sicuramente) e tramite l' invitesSocket, glieli notifica.

Test effettuati:

- login con utente non registrato;
- login con utente registrato, ma password scorretta;
- login con Client già connesso con un altro utente;
- login da parte di un utente connesso;
- login, logout e login successivi
- login senza ricezione di inviti pendenti;
- login con ricezione di inviti pendenti.

6.3 Logout

Il logout di un utente viene effettuato andando a verificare se con il Client richiedente, è connesso un utente (esiste, cioè, una coppia ((clientSocket), username) nella *hash_onlines*) ed in caso affermativo si procede a rimuoverla (altrimenti significa che il Client sta chiedendo la disconessione, senza che prima qualche utente abbia fatto il login).

Test Effettuati:

1. logout da parte di un utente non connesso;
2. logout di un utente connesso.

6.4 Create

La creazione di un documento avviene controllando che l'utente che richiede l'operazione sia connesso, che il documento che richiede di creare non sia già esistente, generando un indirizzo di multicast da associare al documento e creando l'entry (nome_documento, istanza di Document) da inserire nella hash_documents.

Test Effettuati:

1. creazione di un documento da parte di un utente non connesso;
2. creazione di un documento già esistente;
3. creazione di un documento non esistente.

6.5 Share

L'invito alla collaborazione avviene controllando che l'utente che richiede l'operazione sia connesso, che il documento che vuole condividere esista, che sia stato lui a crearlo, che il destinatario dell'invito sia un utente registrato alla piattaforma, che il destinatario non sia già collaboratore del documento in questione e tale invito deve essere notificato, in tempo reale, se il destinatario è connesso (tramite l' invitesSocket associato al SocketChannel con cui il destinatario è connesso e reperibile nella hash_invites), oppure inserito nel suo insieme di inviti pendenti (reperibile nella sua istanza User della hash_users) ed inviatogli quando effettuerà il LOGIN.

Test effettuati:

- tentativo di invito da parte di un utente non connesso;
- invio di invito di un documento non esistente;
- invio di invito di un documento di cui non si è creatore;
- invio di invito a un destinatario non esistente;
- invio di invito ad un destinatario che è già collaboratore;
- invio di invito ad un destinatario online (invito viene inviato subito tramite l'*invitesSocket* associato al SocketChannel con cui quest'ultimo è connesso);
- invio di un invito ad un destinatario offline (invito gli viene notificato quando fa il LOGIN).

6.6 List

La visione della lista dei documenti avviene controllando che l'utente che richiede l'operazione sia connesso, andando ad iterare la lista dei documenti modificabili dall'utente (reperibile nella sua istanza *User* nella hash_users) e per ogni documento, andando a reperire il creatore e gli altri (eventuali) collaboratori (dall'istanza *Document* presente nella hash_documents ed acceduta tramite il nome del documento presente nell'insieme dell'utente). Le varie informazioni così estrappolte vengono memorizzate in una stringa, che verrà notificata come responso al Client. Se l'utente non ha creato nessun documento e non è stato invitato a collaborare da nessuno, viene spedito un messaggio opportuno.

Test Effettuati:

1. visualizzazione della lista dei documenti documenti da parte di un utente non connesso;
2. visualizzazione della lista dei documenti di un utente che non ha creato documenti né è stato invitato a collaborare da nessuno;
3. visualizzazione della lista dei documenti di un utente che ha solo creato documenti o/e è solo stato invitato a collaborare senza mai crearne;

6.7 Show document

La visione di un documento avviene controllando che l'utente che richiede l'operazione sia connesso, che il documento esista e che l'utente sia suo creatore/colaboratore. Per implementare l'invio di un documento (insieme di files), il Server notifica al Client il numero di sezioni che il documento richiesto possiede, per farlo predisporre alla lettura di tale quantitativo di files. Successivamente, spedisce il contenuto delle sezioni del documento (leggendoli dalla directory "/Turing_database/", in mutua esclusione con gli altri workers, tramite l'ausilio si FileLocks). Il Client legge il contenuto e per ogni sezione crea un file opportuno (inserito nella cartella che prende nome dal documento richiesto (cartella creata, eventualmente, se non già esistente)) all'interno della cartella a lui dedicata, nella directory "/Turing_downloads/". Come messaggio finale, una volta inviato il contenuto di tutti i files, il Server notifica al Client quali utenti, eventualmente, stanno editando le sezioni del documento.

Test Effettuati:

1. visualizzazione di un documento da parte di un utente non connesso;
2. visualizzazione di un documento non esistente;
3. visualizzazione di un documento di cui l'utente non è creatore/collaboratore;
4. visualizzazione di un documento, mentre nessuno sta editando le sue sezioni;
5. visualizzazione di un documento, mentre qualcuno sta editando le sue sezioni (viene notificato chi sta editando una data sezione);

6.8 Show section

L'operazione di visione di una sezione non differisce da quella descritta per la visione di un documento, con l'aggiunta del controllo sul numero della sezione (ossia che appartenga al documento desiderato) e l'invio del solo contentuo del file/sezione. Come per l'invio di un documento, il Server notifica al Client quale utente, eventualmente, sta editando la sezione.

Test effettuati:

1. visualizzazione di una sezione di un documento da parte di un utente non connesso;
2. visualizzazione di una sezione di un documento non esistente;
3. visualizzazione di una sezione non appartenente al documento richiesto;
4. visualizzazione di una sezione di un documento di cui l'utente non è creatore/collaboratore;
5. visualizzazione di un documento, mentre nessuno sta editando le sue sezioni;
6. visualizzazione di una sezione di un documento, mentre nessuno la sta editando;

7. visualizzazione di una sezione di un documento, mentre qualcuno la sta editando (viene notificato chi la sta editando).

6.9 Edit

La modifica di un documento avviene controllando che l'utente che richiede l'operazione sia connesso, che il documento richiesto esista, che la sezione richiesta appartenga al documento, che l'utente abbia i permessi per editarlo (sia suo creatore/collaboratore) e che la sezione non sia editata da qualcun'altro. Superati questi controlli, il Server conferisce la mutua esclusione alla sezione, andando ad inserire l'username dell'utente nello slot dell'array delle sezioni contenuto nell'istanza Document del documento richiesto (il conferimento della mutua esclusione avviene tramite un metodo synchronized su tale array) e si segna nell'istanza User dell'utente, quale documento e sezione egli sta editando. Fatto questo il Server invia il contenuto della sezione all'utente (leggendolo dalla directory "/Turing_database/", in mutua esclusione con gli altri workers, tramite l'ausilio si FileLocks)), affinché il Client crei (all'interno della sua cartella dedicata, nella directory "/Turing_edit_mode/") un file con tale contenuto e inizi a modificarlo. Inoltre, tale file viene aperto tramie *java.awt.Desktop* per consentirne una modifica agevole. Successivamente il Server notifica al Client l'indirizzo di multicast associato al documento, per consentirgli di attivare il thread ClientChatListener e rimanere in ascolto del sopraggiungere dei messaggi sulla chat, attraverso l'apertura di un MulticastSocket (registrato all'multicast group notificato). Quando sopraggiunge un messaggio, esso viene inserito dall'ClientChatListener all'interno di una sorta di history, interpellabile dall'utente tramite il comando RECEIVE. Lato Server, infine, ci si occupa di inviare un messaggo in multicast, per notificare agli altri Clients che stanno modificanto il documento, che il corrente utente si è unito alla chat.

Test Effettuati:

1. edit di una sezione di un documento da parte di un utente non connesso;
2. edit di una sezione di un documento non esistente;
3. edit di una sezione non appartenente al documento richiesto;
4. edit di una sezione di un documento di cui l'utente non è creatore/collaboratore;
5. edit di una sezione di un documento già editata da qualcuno (viene notificato chi la sta editando);
6. edit di una sezione di un documento non editata da nessuno.

6.10 End Edit

La conclusione della modifica di un documento avviene controllando che l'utente che richiede l'operazione sia connesso, che il documento richiesto esista, che la sezione richiesta appartenga al documento, che l'utente abbia i permessi per editarlo (sia suo creatore/collaboratore). Superati questi controlli, si verifica se la sezione in questione sia settata per l'editing (altirmenti si invia un messaggio di errore) e se sì, chi sia l'utente che l'ha acquisita. Se chi l'ha acquisita non

è l'utente che ne richiede la conclusione della modifica, viene inviato un messaggio d'errore che contiene il nominativo di chi realmente detiene tale sezione, altrimenti viene inviato una risposta di buon esito al Client, notificandogli che il Server attende l'invio della sezione modificata, per aggiornare il database (la scrittura sul file rispettivo della directory "/Turing_database/" viene fatta in mutua esclusione tramite FileLock, per vigilarne l'accesso tra gli workers che vi leggono e quelli che vi scrivono). Infine, il Client disattiva il thread ClientChatListener ed il Server si occupa di inviare un messaggo in multicast, per notificare agli altri Clients che stanno modificanto il documento, che il corrente utente è uscito dalla chat.

Test Effettuati:

1. end-edit di una sezione di un documento da parte di un utente non connesso;
2. end-edit di una sezione di un documento non esistente;
3. end-edit di una sezione non appartenente al documento richiesto;
4. end-edit di una sezione di cui l'utente non è creatore/collaboratore;
5. end-edit di una sezione di un documento di cui si è creatori/collaboratori, ma non è settata per l'edit;
6. end-edit di una sezione di un documento di cui si è creatori/collaboratori, ma è settata per l'edit da un altro utente (viene notificato chi realmente la sta editando);
7. end-edit di una sezione di un documento di cui si è creatori/collaboratori e si è fatto l'edit.

6.11 Send

L'invio di un messaggio sulla chat avviene controllando che l'utente che richiede l'operazione sia connesso e che stia editando la sezione di un documento. I controlli vengono fatti lato Client, in quanto il Client è in possesso dei seguenti riferimenti:

- username dell'utente connesso con il Client;
- nome del documento e della sezione, eventualmente, da lui editate;

L'invio di un messaggio, per quanto spiegato precedentemente, passa per il Server e di conseguenza è il Server che tramite un synchronized statement sul DatagramSocket, assicura il corretto invio del messaggio in multicast da parte del Client.

6.12 Receive

La lettura dei messaggi della chat avviene controllando che l'utente che richiede l'operazione sia connesso e che stia editando la sezione di un documento. I controlli vengono fatti lato Client, in quanto il Client è in possesso dei seguenti riferimenti:

- username dell'utente connesso con il Client;
- nome del documento e della sezione, eventualmente, da lui editate;
- *history* dove vengono memorizzati i messaggi ricevuti, ma non ancora visionati, presente nel thread ClientChatListener.

Quando l'utente richiede questa operazione, vengono prelevati uno ad uno i messaggi e stampati, per visionarli. Siccome nel frattempo potrebbero giungere altre messaggi sulla chat ed il thread ClientChatListener, di conseguenza, inserirli mentre il Client li rimuove, la coda dei messaggi è una BlockingQueue, per mantenerne la consistenza.

Test effettuati:

1. receive con utente non connesso;
2. receive con utente connesso, ma che non sta editando alcun documento;
3. receive con utente connesso, che sta editando un documento ma non ha ricevuto messaggi;
4. receive con utente connesso, che sta editando un documento e ha ricevuto messaggi;

6.13 Help

In caso si abbiano dubbi sulla sintassi dei comandi da sottomettere al sistema, digitare il seguente comando per avere una panoramica delle operazioni e degli argomenti necessari:

$$turing\ --help$$

6.14 Exit

Comando aggiuntivo, che tramite la seguente sintassi, permette di terminare il Client istantaneamente (a meno di non essere in editing-mode. In tal caso, viene chiesto di ultimare l'editazione):

$$turing\, exit$$

7 Note

1. Per compilare ed eseguire TURING, è necessario posizionarsi all'interno della cartella */src/* del progetto e lanciare da linea di comando:
   - per compilare ed avviare il Server:

     $$javac\, TuringServer.java$$

     $$java\, TuringServer$$

   - per compilare ed avviare uno o più Clients:

     $$javac\, TuringClient.java$$

     $$java\, TuringClient$$

2. È necessario installare i moduli *libcanberra-gtk-module* e *libcanberra-gtk3-module* su Linux, altrimenti quando si manda la richiesta di fare l'editing di un documento e il documento viene aperto per consentirne la modifica (l'apertura viene fatta tramite *java.awt.Desktop*), potrebbe comparire il seguente messaggio (che in ogni caso non ostacola il proseguimento e la correttezza dell'esecuzione):

   $$Gtk - Message : Failed\ to\ load\ module\ "canberra - gtk - module"$$

   Con un semplice:

   $$sudo\ apt\ install\ libcanberra - gtk - modulelibcanberra - gtk3 - module$$

   l'eventuale messaggio scompare.

3. In tutto il codice sono abbondanti i commenti e le spiegazione dei passaggi più delicati. Riferirsi ad esso se ci sono dei punti poco chiari riguardo a quando esposto.