UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Relazione del progetto

disTribUted collaboRative edItiNG (TURING)

Matteo Biviano
Matricola 543933

Corso di Laurea: **Informatica**
Insegnamento: **Laboratorio di Reti di Calcolatori**
Docente: **Federica Paganelli**
A.A. 2018/2019

INDICE

I INTRODUZIONE

Il documento si pone l'obiettivo di esporre e fornire al lettore una descrizione esaustiva del progetto e della sua implementazione, nonché di altre possibili implementazioni. Si rende noto che il codice del progetto è stato adeguatamente commentato per guidare l'utente nell'implementazione.
Il progetto **TURING** è uno strumento utilizzato per l'editing di documenti in modo collaborativo fra più utenti. È stato realizzato con un'interfaccia grafica, rendendo il testing del progetto più semplice e veloce.
L'interfaccia grafica è realizzata interamente in vettoriale attraverso l'uso di **Adobe Illustrator** esportando le immagini (successivamente caricate nel progetto) in formato **.png** (Portable Network Graphics). Il formato .png infatti è un formato **lossless**, la cui compressione non implica quindi perdita di dati.
Si puntualizza, inoltre, che non è stata gestita la persistenza dei dati tra un'esecuzione e l'altra del progetto, in quanto non richiesto nelle specifiche del progetto.

1. ARCHITETTURA

Di seguito, verranno illustrate l'architettura del progetto e le principali scelte progettuali effettuate per la realizzazione.

1.1 Server – Scelte progettuali (IO vs NIO)

Per la realizzazione dell'architettura del server sono state presentate durante il corso di "Laboratorio di Reti di Calcolatori" due possibili soluzioni:

1) Gestione del server multithreaded sincrona, attraverso l'utilizzo di I/O bloccante (usando **Sockets** della libreria **IO**);

2) Gestione del server che effettua il multiplexing dei canali, quindi una gestione monothread sincrona con I/O non bloccante (attraverso l'uso dei **Selectors** della libreria **NIO**).
Dovendo scegliere tra velocità e scalabilità; differenze principali nel trade-off tra le due possibilità **java.io** e **java.nio**, nell'implementazione fornita si è optato per la prima soluzione. Poiché sotto carichi non eccessivi, quali si presuppone siano quelli del progetto didattico realizzato, la prima opzione garantisce una maggiore velocità.

1.1.1 Java IO

La libreria **java.io** si basa su una gestione **bloccante** delle operazioni di I/O. La libreria offre un'insieme di astrazioni per la gestione dell'I/O basandosi sul concetto di **stream**, il quale è una sequenza di informazioni di lunghezza illimitata. Gli streams sono caratterizzati dal fatto di essere **one way**, se un programma ha bisogno di dati sia in input che in output è necessario utilizzare due stream separati. Il comportamento **bloccante** degli streams si realizza nelle richieste di scrittura o lettura. Un'applicazione che legge un dato dallo stream o lo scrive si blocca finché i dati non sono stati completamente letti o completamente scritti.

1.1.2 Java NIO

La libreria **java.nio** permette, rispetto alla versione **java.io**, un approccio I/O buffer oriented. Nella versione precedente infatti, non ci si poteva muovere liberamente all'interno dello stream di dati. In questa versione, i dati sono letti in un buffer da dove è possibile, successivamente, processarli. Inoltre, è possibile iterarli a piacimento concedendo una maggiore flessibilità durante la fase di elaborazione. Tuttavia, bisogna controllare che il buffer contenga tutti i dati richiesti prima di processarli e che non vi siano dati non ancora elaborati nel buffer, prima di sovrascriverli.

La modalità **non bloccante** di **java.nio** permette all'applicazione di richiedere dati da un canale e ottenere solo quelli che sono attualmente disponibili al suo interno, invece di rimanere bloccati fino a che i dati non siano resi disponibili (come invece impone di fare la versione java.io).

1.1.2.3 Server NIO

Questa parte del documento (unico caso in cui viene riportato del codice nel documento) si pone ulteriormente l'obiettivo di dare una descrizione di massima di come sarebbe possibile implementare un server per il progetto utilizzando la libreria **java.nio**.

```
public class Server {
    public static int serverPort = 3029;
    public static void main( String[] args ) {
       // Canale selezionabile per l'ascolto dei socket
       ServerSocketChannel serverSocketChannel = null;
       // Il selettore che effettua il multiplexing dei canali selezionabili
       Selector selettore = null;
       // La registrazione di un canale selezionabile con un selettore è
       // rappresentata dall'oggetto SelectionKey
       /**
        * Un selettore mantiene 3 possibili set di selectionKey:
        * - Il key set che contiene le chiavi che rappresentano il canale
        *   di registrazione al selettore
        * - Il selected-key set che contiene le chiavi per far capire che si
        *   è pronti ad effettuare operazioni
        * - Il cancelled-key set che contiene le chiavi cancellate, ma i cui
        *   canali non sono ancora stati deregistrati
        */
       SelectionKey selectionKey;
       try {
           // Il server socket channel viene creato attraverso l'invocazione
           // del metodo "open()" della medesima classe; quindi non è possibile
           // crearlo da un ServerSocket preesistente
           serverSocketChannel = ServerSocketChannel.open();
           // Attraverso il metodo "socket()" restituiamo il server socket
           // associato con questo canale per poi effetuare il "bind" con l'indirizzo
           serverSocketChannel.socket().bind(new InetSocketAddress(serverPort));
           // Configuriamo la modalità non bloccante del servizio
           serverSocketChannel.configureBlocking(false);
           // Creiamo il selettore invocando il medoto "open()" della stessa classe
           // NOTA: questa creazione può essere personalizzata attraverso il metodo
           // "openSelector" restituendo quindi un selettore personalizzato
           selettore = Selector.open();
           // Registriamo il canale al selettore specificato, restituiendo una selectionKey
           selectionKey = serverSocketChannel.register(selettore, SelectionKey.OP_ACCEPT, null);
       } catch(IOException e) {
         e.printStackTrace();
       }
          Set<SelectionKey> setKey;
          // Iteratore per le chiavi
          Iterator<SelectionKey> iterator;
          try {
              // Selezioniamo il set di chiavi i cui canali sono pronti per operazioni di I/O
              // Il metodo esegue una operazione di selezione bloccante
              int keysIO = selettore.select();
              // Acquisiamo il set di chiavi selezionate dal selettore
              setKey = selettore.selectedKeys();
              // Acquisiamo l'iteratore per il set di chiavi
              iterator = setKey.iterator();
              // Registriamo il canale con il selettore specificato, restituendo una selectionKey
              selectionKey = serverSocketChannel.register(selettore, SelectionKey.OP_ACCEPT, null);
          } catch(IOException e) {
              e.printStackTrace();
              return;
          }
```

```
            // Iteriamo attraverso l'iteratore sul set di chiavi
            while(iterator.hasNext()) {
                // Acquisiamo la chiave corrente
                SelectionKey key = iterator.next();
                // In base al tipo di chiave effettuiamo un'operazione diversa
                if(key.isAcceptable()) {
                    // Un client vuole connettersi
                    try {
                        // Accettiamo il socket che descrive il client
                        SocketChannel skt_client = serverSocketChannel.accept();
                        if(skt_client == null) continue;
                        // Modalità bloccante
                        skt_client.configureBlocking(false);
                        // Registriamo il canale con il selettore specificato
                        skt_client.register(selettore, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
                    } catch(IOException ex) {
                    }
                } else if(key.isReadable()) {

                // Un client richiede un'operazione
                // Acquisiamo il socket che descrive il client
                SocketChannel clientChannel = (SocketChannel)key.channel();
                /****
                 *
                 * Gestiamo l'operazione richiesta
                 *
                 ****/
            }
            iterator.remove();
        }
    }
}
```

1.2 Server – Architettura

Come ogni paradigma di tipo Client-Server, il server è quel processo che deve essere attivo prima dell'esecuzione dei client. Il server del progetto è realizzato nella classe **TuringServer** ed è strutturato in più fasi, descritte di seguito.

1.2.1 Fase 1 – Caricamento delle configurazioni

In questa fase è stata prevista la possibilità di modificare le configurazioni globali, utilizzate all'interno dell'implementazione, attraverso la lettura di un file di configurazione presente in **DATA/turingServer.conf**. Se al momento dell'esecuzione del server non si specifica questo file tra gli argomenti, il server utilizzerà le configurazioni di default previste al momento dell'implementa-zione. In entrambi i casi, verranno mostrate attraverso un **JOptionPane** le configura-zioni con cui si sta eseguendo il server, come mostrato in Figura 1.

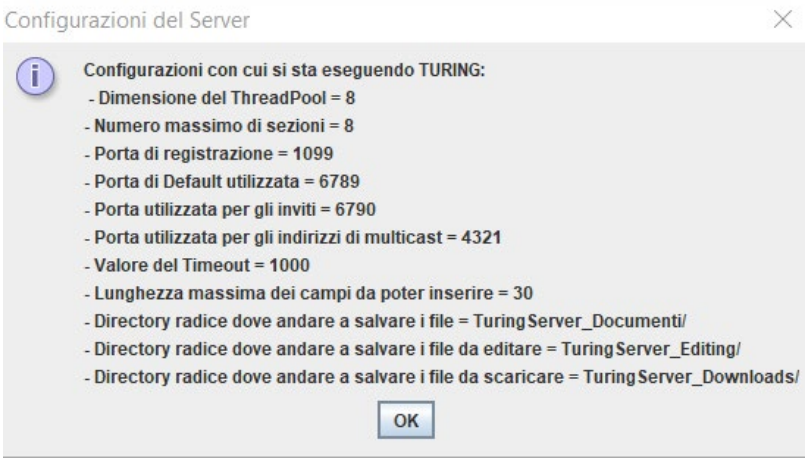

Figura 1 - MessageDialog delle configurazioni usate

1.2.2 Fase 2 – Inizializzazione delle strutture

Dopo la fase di caricamento delle configurazioni, il server inizializzerà il **ServerSocket**, il **ThreadPool** e tutte le strutture dati utilizzate all'interno della classe **ServerPayload**.

1.2.3 Fase 3 – Attivazione del servizio di registrazione

Il server, al momento della sua attivazione, mette a disposizione un servizio di registrazione tramite stub RMI, al quale il client può accedere per effettuare una **Remote Invocation** del metodo **registrationHandler**, al fine di registrare un nuovo utente (descritto da **username** e **password**).

1.2.4 Fase 4 – Creazione della directory

Poiché il progetto non prevede una gestione della consistenza delle informazioni tra un'esecuzione e l'altra del server, in questa fase si crea la directory dove memorizzare i documenti gestiti dal servizio TURING, eliminando tutte le cartelle e i file frutto di un esecuzione precedente.

1.2.5 Fase 5 – Installazione listener

In questa fase, il server si occupa di instanziare e attivare il thread listener che gestisce il servizio degli inviti in diretta (in live), di cui si parlerà successivamente.

1.2.6 Fase 6 – Ciclo di attesa

Il server, finite le inizializzazioni, diventa il thread Listener del progetto, attivando un ciclo infinito di attesa di richieste di connessioni da parte di clients. Ogni connessione dopo essere stata accettata viene processata dal **ThreadPool**, in modo tale da non impedire ad altri clients di potersi a loro volta connettere.

2. CLASSI PRINCIPALI DEL PACKAGE "TuringServer"

2.1 Classe Utente (Utente.java)

La classe Utente rappresenta un utente iscritto al servizio TURING. Un utente è composto dai seguenti gruppi di campi:

1) Dati di contesto associati ad un utente che rappresentano i tre insiemi (**Set**) di supporto che servono per gestire gli utenti e le loro richieste

- **set_doc**: è l'insieme dei documenti che un utente può modificare (poiché creatore o collaboratore)
- **set_pendingDocs**: è l'insieme dei documenti ai quali l'utente è stato invitato, per la collaborazione, mentre era offline. Questo insieme, attraverso l'apposito metodo **sendInformationToClient()** (presente nella classe **ConnectionHandler**), verrà inviato all'utente ogni qual volta effettua un nuovo login.
- **set_liveDocs**: è l'insieme dei documenti che l'utente è stato invitato a modificare, mentre era collegato al servizio. Attraverso l'apposito Thread, questo insieme verrà costantemente notificato all'utente specifico attraverso un **pop-up**.

2) Dati identificativi dell'utente, che rappresentano le due **Stringhe** utilizzate per identificare un utente sia in fase di registrazione sia durante tutta la sua permanenza nel servizio

- **username**: nome dell'utente, univoco all'interno del database degli utenti
- **password**: chiave di sicurezza associata allo specifico utente.

La classe contiene, inoltre, dei metodi per modificare ed effettuare operazioni utilizzando i campi appena esposti.

2.2 Classe Documento (Documento.java)

La classe Documento rappresenta un documento registrato al servizio TURING. Un Documento è composto dai seguenti gruppi di campi:

1) Dati di contesto del documento:

- **users**: Insieme (**Set**) di utenti invitati a modificare quel documento
- **reentrantLocks**: **Array** di **ReentrantLocks** a cui ogni posizione corrisponde una sezione del documento (quindi un file **.txt**). Questo array è utilizzato per implementare la mutua-esclusione nelle modifiche delle sezioni di un documento, in quanto ogni sezione non può essere modificata da più di un utente nello stesso momento.
- **chat**: rappresenta un **InetAddress**, cioè un indirizzo di multicast associato ad un documento al momento della sua creazione. Attraverso questo indirizzo sarà possibile agli utenti, collaboratori del documento, di poter chattare (tramite apposita inter-faccia) durante la modifica di una sezione del documento. Al momento della creazione dell'indirizzo si utilizza un range che va da 225.0.0.0 fino a 239.255.255.255 poiché ritenuto sufficientemente grande per i fini del progetto. Alternativamente, si sarebbe potuto estendere la creazione degli indirizzi partendo da 224.0.1.0, in modo da utilizzare l'intero range disponibile.

2) Dati identificativi del documento

- **documento**: Stringa che rappresenta il nome del documento. Anche questo, come il nome di un utente, deve essere univoco all'interno del servizio
- **utente**: Stringa che identifica il nome del creatore del documento. La classe contiene, inoltre, dei metodi per modificare ed effettuare operazioni utilizzando i campi appena esposti.

2.3 Classe ServerPayload (ServerPayload.java)

La classe ServerPayload rappresenta il contenitore dei dati di contesto della classe TuringServer, in quanto contiene tutte le istanze utili per la realizzazione del servizio e i metodi per utilizzarle. In particolare, contiene i seguenti campi:

- **onUser**: Insieme (**Set**) degli utenti attualmente connessi al servizio,
- **offUser**: Insieme (**Set**) degli utenti attualmente non connessi al servizio.

Gli insiemi appena descritti sono utilizzati per gestire correttamente gli inviti (live e non) e le connessioni e disconnessioni al servizio.

- **indirizzi**: Insieme (**Set**) di indirizzi di multicast per le chat dei documenti. L'insieme è utilizzato in particolar modo per associare ad ogni documento un indirizzo univoco
- **hash_user**: Tabella hash (**ConcurrentHashMap<String, Utente>**) che rappresenta il database degli utenti registrati e realizza l'associazione <nome_utente> - <utente>
- **hash_document**: Tabella hash (**ConcurrentHashMap<String, Documento>**) che rappresenta il database dei documenti registrati e realizza l'associazione <nome_documento> - <documento>
- **lockHash**: Oggetto (**Object**) utilizzato all'interno del progetto per gestire la sincronizzazione tra il database degli utenti e il database dei documenti
- **lockUser**: Oggetto (**Object**) utilizzato all'interno del progetto per gestire la sincronizzazione tra l'insieme degli utenti online e l'insieme degli utenti offline

2.4 Classi Inviti Live (ListenerInvites.java e InvitesTask.java)

La classe **ListenerInvites** rappresenta il thread sender che viene attivato per gestire le richieste di inviti live, cioè gli inviti a collaborazione quando l'utente ricevente è online. Questo thread attende su una **listenerServerSocket** nuove connessioni che verranno richieste ogni volta che un utente si connette al servizio TURING. Ogni volta che arriva una connessione, come specificato in precedenza per il server, questo thread effettua la gestione delle richieste utilizzando un ThreadPool che attiva il rispettivo Runnable **InvitesTask**.

La classe InvitesTask acquisisce tramite il **BufferedReader** di input il nome utente che ha richiesto il servizio di inviti live (cioè l'utente da servire), in modo tale da accedere all'insieme degli inviti live di questo utente e notificarli costantemente, attraverso il **DataOutputStream** di output.
Lato client il ricevente di inviti live, è rappresentato dal thread InvitesHandler (**InvitesHandler.java**), il quale non ha interfaccia grafica, poiché permette al servizio di inviti live di esistere in un ottica client-server.

2.5 Classe ConnectionHandler (ConnectionHandler.java)

La classe ConnectionHandler si occupa di implementare le operazioni di richiesta da ogni client connesso al servizio TURING. Questa classe rappresenta il Runnable che viene creato quando il server accetta una nuova connessione e viene eseguito dal Thread-Pool corrispondente. In questa classe verrà attivato un ciclo di attesa in cui si legge l'operazione richiesta, tramite il **BufferedReader** di input, e si gestisce attraverso l'apposito metodo **switchRequest(operazione)**. Nella fase di gestione dell'operazione, il server invierà i risultati al client implementando un semplice protocollo TCP. Per mantenere la consistenza dei dati all'interno di una esecuzione del servizio, se il client interrompe la sua esecuzione (volutamente o per crash) si effettua una fase di chiusura dei canali e liberazione delle risorse occupate, tra cui la sezione eventualmen-te bloccata durante una modifica.

3 PRINCIPALI STRUTTURE DATI – MOTIVAZIONI

3.1 LinkedHashSet

All'interno del progetto sono stati utilizzati dei **Set**, poiché le informazioni contenute in essi devono essere univoce all'interno del servizio TURING, ed infatti i Set non possono contenere duplicati. Tuttavia, poiché il normale Set non fornisce garanzie su come gli elementi siano ordinati, è stato scelto di usare l'implementazione di **LinkedHashSet**. Il LinkedHashSet risulta più efficiente di HashSet su una grande mole di dati poiché permette di mantenere i dati ordinati al momento dell'inserimento. Ipotizzando invece un uso più didattico del progetto, quindi con una mole di dati molto bassa, sarebbe stato più utile utilizzare un **HashSet** che, non dovendo ordinare i dati, risulta più veloce negli inserimenti, rispetto a LinkedHashSet. Per le stesse motivazioni di LinkedHashSet si sarebbe potuto utilizzare **TreeSet**, specificando il comparatore attraverso il quale ordinare i dati. Sarebbe stato, inoltre, possibile utilizzare **Liste**, controllando prima dell'inserimento se il dato è già presente, inserendolo nella struttura solo in caso negativo.

3.2 ConcurrentHashMap

Per la gestione delle informazioni sensibili del progetto, il server utilizza due ConcurrentHashMap come database. Queste HashMap essendo Concurrent, comportano che le operazioni siano Thread-safe in un contesto in cui più utenti lavorano in cooperativa nello stesso istante. Tuttavia, una **ConcurrentHashMap** non assicura con meccanismi interni che una Write e una Read siano coerenti, in quanto l'operazione Read (retrieval operation) riporta i risultati dell'ultima operazione terminata. Per questioni di efficienza e scalabilità non sono state utilizzate delle **Synchronized-Map** poiché, mentre una ConcurrentHashMap blocca solo una parte dei dati che vengono aggiornati, lasciandone il resto accessibili ad altri thread; la SynchronizedMap blocca tutti i dati durante l'aggiornamento ed eventuali altri thread possono accedere ai dati solo quando viene rilasciato il blocco. Inoltre, mentre una ConcurrentHashMap può garantire l'assenza di **ConcurrentModificationException** quando un thread aggiorna la mappa e un altro la attraversa con l'iteratore, questo non è garantito in una SynchronizedMap.

3.3 Object

Per la sincronizzazione dei due database e dei due insieme di utenti (online e offline), come già detto sono stati utilizzati due oggetti. Questi due oggetti sono stati preferiti rispetto a lock esplicite sia per utilizzare un altro argomento presentato durante il corso, sia perché permettono di sincronizzare blocchi di codice anziché metodi interi, consentendo di avere sezioni critiche di dimensioni inferiori rispetto all'intero metodo.

3.4 ThreadPool

Il ThreadPoolExecutor utilizzato dal server per gestire le connessioni (come anche quello utilizzato nel thread che gestisce gli inviti Live) è un **FixedThreadPool**. Non è stato utilizzato un **CachedThreadPool** in quanto, in caso di un carico maggiore, senza una limitazione nella coda delle attività il CachedThreadPool potrebbe causare una latenza elevata o memoria insufficiente o addirittura causare dei problemi nella creazione dei Thread. Tuttavia, ipotizzando l'uso didattico del progetto con un numero di client basso anche la soluzione CachedThreadPool sarebbe stata possibile.

4 CLIENT – INTERFACCIA E OPERAZIONI

Come già specificato nell'introduzione, è stato scelto di realizzare il client utilizzando un'interfaccia grafica e la libreria **java.swing**. Di seguito, vengono riportate sinteticamente le varie interfacce previste.

4.1 Interfaccia di Registrazione e Login (TuringClient.java)

Eseguendo il client **TuringClient**, apparirà (come detto in precedenza per il server) un pop-up che elenca le configurazioni con cui si sta eseguendo il client. Successivamente si aprirà la seguente l'interfaccia di "Registrazione e Login" del servizio TURING, con i campi dove poter inserire il nome utente e la password.

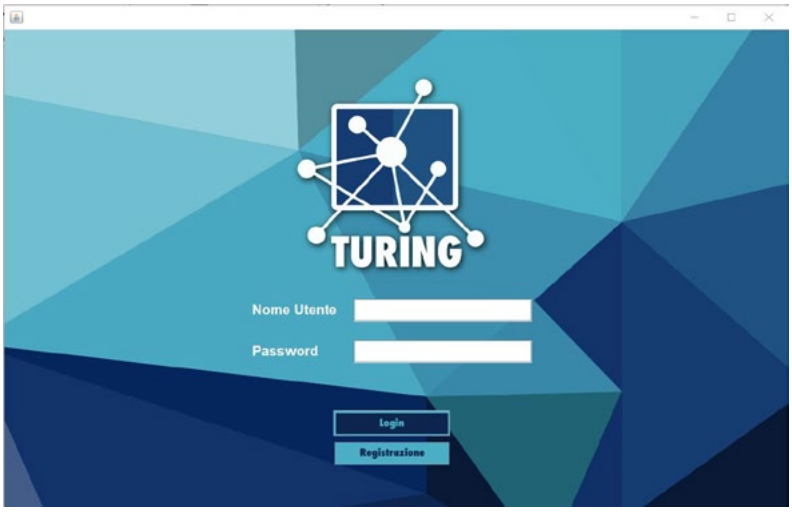

Figura 2 – Interfaccia di "Registrazione e Login"

Se il server non è online al momento dell'esecuzione del client, oppure se al momento della registrazione o del login il server è offline, verrà mostrato il seguente pop-up che informerà l'utente della situazione e l'applicazione verrà chiusa.

Figura 3 – Messaggio di errore "Server Offline"

Non sono stati effettuati controlli sulla tipologia dei caratteri inseribili come "nome utente" e "password", tuttavia se non compilati entrambi i campi verrà notificato l'errore al client attraverso il pop-up.

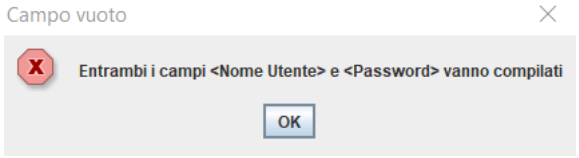

Figura 4 – Messaggio di errore "Campo vuoto"

4.1.1 Registrazione

La gestione della registrazione è implementata tramite RMI, il quale rende possibile l'invocazione di metodi originariamente del server, da remoto. Il server RMI crea l'oggetto remoto e lo passa come stub al **Registry**. Uno stub permette al client di usare i metodi definiti lato server come remoti, inoltrandoli all'oggetto remoto originale.
Nel caso in cui si tentasse di registrare un utente già registrato al servizio, verrà notificato l'errore al client tramite il pop-up, viceversa verrà notificato il successo dell'operazione.

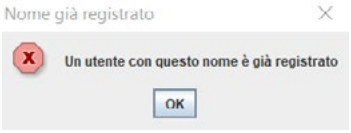

Figura 5 – Messaggio di errore "Nome già registrato"

Figura 6 – Messaggio di successo "Utente registrato"

4.1.2 Login

Il login di un utente è implementato tramite una connessione TCP. Una volta che la richiesta ha avuto successo, l'username del client verrà rimosso dal Set degli utenti offline e inserito nel Set degli utenti online. Successivamente, verranno inviati al client gli inviti a collaborazioni ricevuti mentre era offline, il quale attiverà il listener per la ricezione degli inviti in diretta. Se al momento del login l'utente non è registrato, oppure è già online verranno notificati gli errori attraverso gli opportuni popup.

Figura 7,8 – Messaggi di errore "Login non effettuato" e "Utente già connesso"

In caso di successo dell'operazione, si attiverà la schermata successiva (schermata di "Connessione") rilasciando le risorse della schermata attuale.

4.2 Interfaccia di Connessione (ConnectionGUI.java)

Dopo aver effettuato il Login, verrà disabilitata la schermata di "Registrazione e Login" e abilitata la schermata seguente (l'interfaccia è stata realizzata in modo il più possibile chiaro e intuitivo per l'utente).

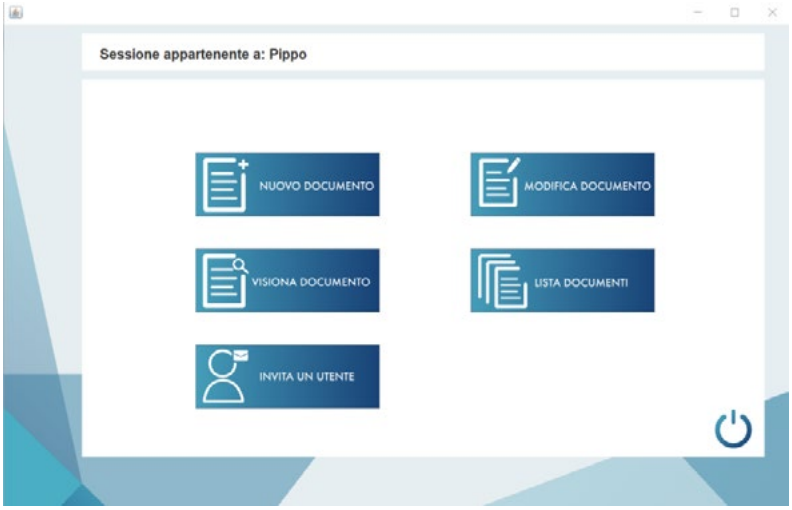

Figura 9 – Schermata di "Connessione"

L'interfaccia presenta in alto l'Username dell'utente a cui appartiene la sessione corrente. Questo Label supporta l'utente in quanto permette di distinguere i vari clients aperti. Come nell'operazione di login anche nelle seguenti operazione è stata utilizzata una connessione TCP verso il server.

4.2.1 Creazione di un nuovo documento

Premendo l'apposito bottone, si aprirà una finestra di **ConfirmDialog**, in cui verrà data la possibilità all'utente di inserire i dati opportuni (Nome del documento, Numero di sezioni del documento).

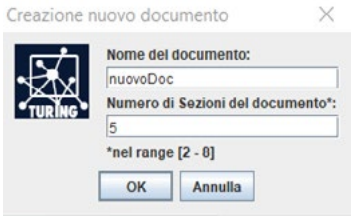

Figura 10 – ConfirmDialog per la "Creazione di un nuovo documento"

Se il nome inserito non è già presente nel database e se il numero di sezioni inserito va da 2 (numero minimo di sezioni per avere editing collaborativo) al numero massimo di sezioni, presente tra le configurazioni di client e server, verrà notificato il successo dell'operazione.

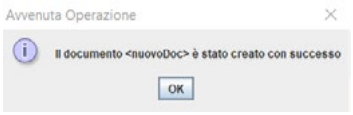

Figura 11 – Messaggio di successo "Documento creato"

In questo modo, all'interno del Path "**/Turing_Matteo_Biviano_543933/Turing Server_Documenti**" (creato durante l'inizializzazione del server) verrà generata una cartella nominata con il nome inserito nel campo "Nome del documento:" che conter-rà un file in formato .txt per ogni sezione indicata nel campo "Numero di Sezioni del documento:".

4.2.1.1 Gestione degli errori

Per la creazione di un nuovo documento sono stati gestiti i seguenti casi di errore, che verranno notificati all'utente attraverso opportuni messaggi di pop-up:

- Inserimento del nome di un documento già presente.

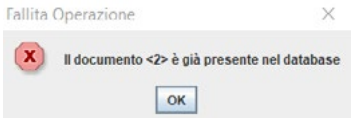

Figura 12 – Messaggio di errore "Documento già presente"

- Inserimento di un numero di sezione minore di 2 o maggiore del numero di sezioni massimo stabilito.

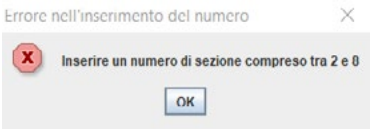

Figura 13 – Messaggio di errore "Sezione non valida"

- Inserimento di lettere nella text box dove deve essere inserito il numero degli utenti.

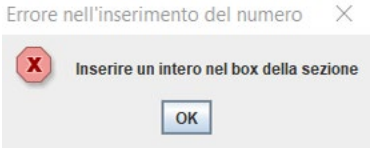

Figura 14 – Messaggio di errore "Carattere non valido"

4.2.2 Invitare un Utente a collaborare

Questo pulsante permette di invitare un utente a collaborare alla modifica di un documento. Premendo il pulsante si aprirà una piccola finestra che richiederà di inserire nome del Documento e l'utente da invitare.

Figura 15 – ConfirmDialog dell'operazione di "Invito all'editing"

Se i campi inseriti risultano corretti, verrà notificata l'avvenuta operazione all'utente tramite l'apposito pop-up.

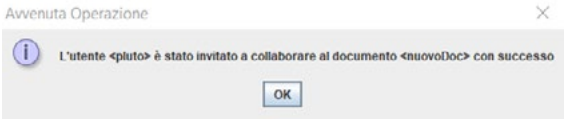

Figura 16 – Messaggio di successo "Utente invitato"

In caso di avvenuta operazione, si avranno due comportamenti distinti a seconda che l'utente ricevente l'invito risulti online o offline:

- Se l'utente ricevente è offline al momento dell'invito, viene inserito il nome del documento all'interno del Set apposito dell'utente.
- Se l'utente ricevente è online al momento dell'invito, viene inserito il nome del documento all'interno del Set apposito dell'utente che, attraverso il Thread Listener gestore degli inviti Live, verrà spedito all'utente. Il ricevente infatti riceverà la seguente notifica.

Figura 17 – Messaggio di notifica di un nuovo invito

4.2.2.1 Gestione degli errori

Sono stati gestiti i seguenti casi di errore:

- Invito alla collaborazione di un utente non registrato.

Figura 18 – Messaggio di errore "Utente non registrato"

- Invito alla collaborazione di un utente registrato ad un documento non presente.

Figura 19 – Messaggio di errore "Documento non presente"

- Invito alla collaborazione di un utente registrato ad un documento di cui non si è creatore.

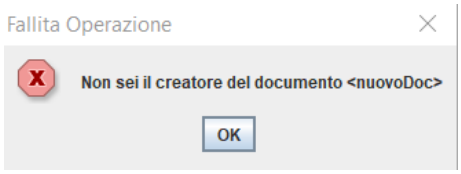

Figura 20 – Messaggio di errore "Utente non creatore del documento"

- Invito di se stesso a collaborare al documento.

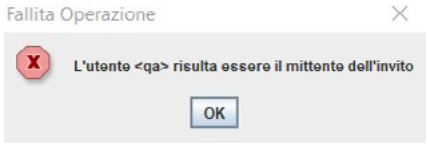

Figura 21 – Messaggio di errore "Mittente e destinatario coincidono"

- Invito di un utente già collaboratore del documento

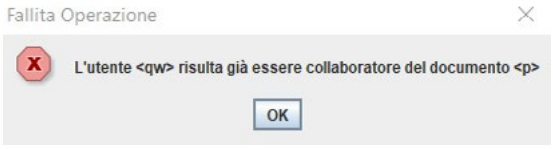

Figura 22 – Messaggio di errore "Utente già collaboratore"

4.2.3 Visione di un documento o di una sezione

Il pulsante permette di inserire il nome del documento e il numero di sezione da voler visionare. Se il numero di sezione inserito è uguale al numero di sezioni massimo consentito dall'implementazione, verrà scaricato il documento completo.

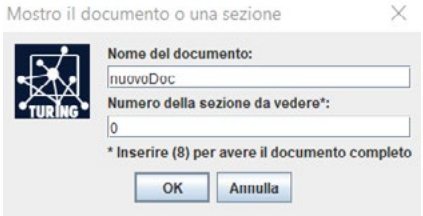

Figura 23 – ConfirmDialog per l'operazione di "Visione di un documento"

Ad operazione avvenuta verrà creato nel Path "**/Turing_Matteo_Biviano_543933/ TuringServer_Downloads**", una nuova cartella (se non presente) con il nome dell'utente che ha richiesto l'operazione e il file desiderato. La notifica di successo dell'operazione cambia a seconda se viene richiesta una sola sezione o il documento per intero:

- Richiesta di una sezione.

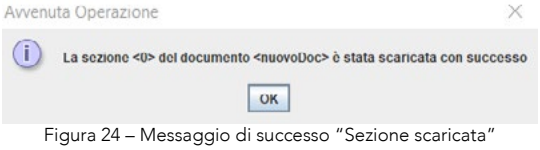

Figura 24 – Messaggio di successo "Sezione scaricata"

- Richiesta di un intero documento.

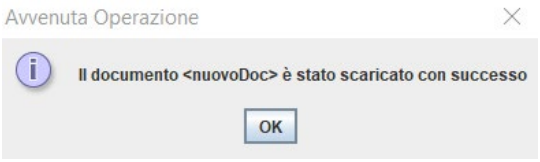

Figura 25 – Messaggio di successo "Documento scaricato"

Il documento, intero o meno, viene passato ad un FileChannel per la lettura, successivamente ad un SocketChannel per inviarlo al client richiedente, per poi ritornare attra-verso il FileChannel nella destinazione specificata in precedenza.

4.2.3.1 Casi speciali – file in editing

Poiché il progetto prevede anche la modifica collaborativa dei documenti, potrebbe presentarsi il caso che un documento richiesto in lettura sia in modifica in alcune sue parti. Questo determina due possibili situazioni:

- Se si richiede una sezione che un altro utente sta modificando, il sistema scaricherà la vecchia versione della sezione, avvisando di conseguenza l'utente.

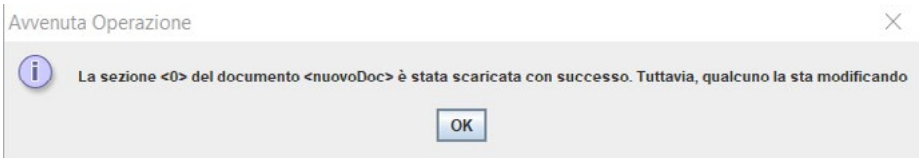

Figura 26 – Messaggio di successo "Sezione occupata scaricata"

- Se si richiede un intero documento, di cui alcune sezione sono in modifica, verrà scaricato la vecchia versione del documento, notificando all'utente l'avvenuta operazione e mostrando la lista delle sezioni che sono in modifica.

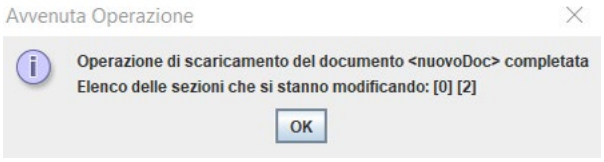

Figura 27 – Messaggio di successo "Documento scaricato"

4.2.3.2 Gestione errori

Sono stati gestiti i seguenti casi di errore:

- Visione di un documento non esistente.

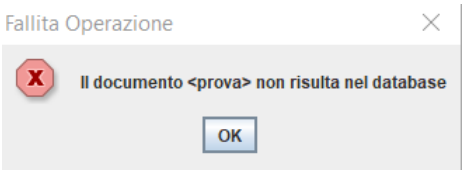

Figura 28 – Messaggio di errore "Documento non presente"

- Visione di un documento di cui non si è collaboratori, oppure visione di una sezione non esistente del documento.

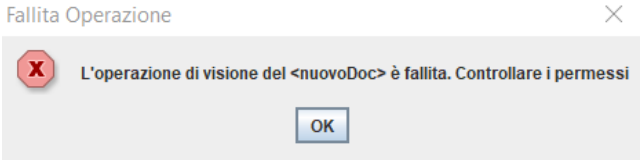

Figura 29 – Messaggio di errore "Visione di un documento fallita"

4.2.4 Visione della Lista dei documenti

Questo pulsante permette la visualizzazione della lista dei documenti che un utente è abilitato a modificare, quindi tutti i documenti di cui è il creatore o ai quali è stato invitato come collaboratore. Il pulsante apre una finestra riassuntiva per tutta la lista di documenti, invece di una finestra per ogni documento, caratteristica che può risultare negativa in quanto, non essendo previsto uno **ScrollPane**, se la lista dei documenti è molto lunga risulterà difficile consultarla per intero.
In caso di esito positivo della operazione, viene richiesto il Set dei documenti che l'utente è abilitato a modificare, scritto poi nell'apposita finestra in maniera schematica.

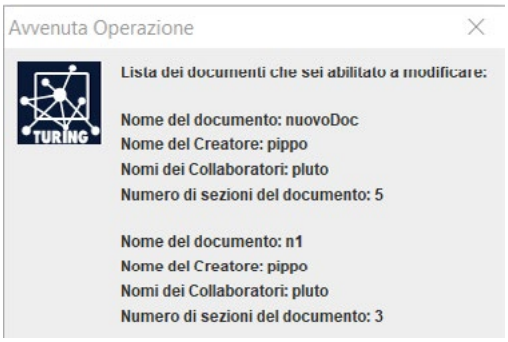

Figura 23 – ConfirmDialog per l'operazione di "Visione di un documento"

4.2.5 Modifica della sezione di un documento

Il pulsante permette di specificare il nome del documento e la sezione da modificare.

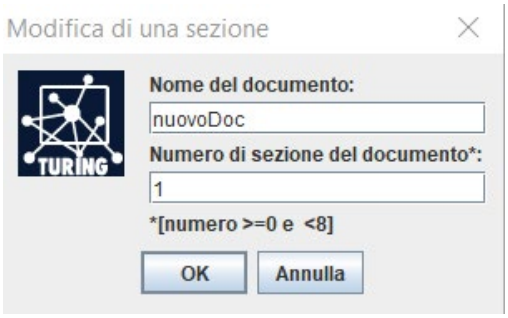

Figura 31 – ConfirmDialog dell'operazione di "Modifica di una sezione"

In caso di esito positivo della richiesta, viene scaricato il file, creando all'interno del Path "**/Turing_Matteo_Biviano_543933/TuringServer_Editing**" una cartella identificata dal nome dell'utente richiedente (se non già presente), inserendovi il file. Successivamente verranno liberate le risorse dell'interfaccia corrente e istanziata l'interfaccia successiva (l'interfaccia di "Editing"). Tutto questo verrà notificato all'utente tramite un apposito pop-up.

Figura 32 – Messaggio di successo "Richiesta di modifica"

4.2.5.1 Gestione degli errori

Per la richiesta di modifica della sezione di un documento sono stati attenzionati i seguenti casi di errore, opportunamente notificati all'utente:
- Modifica di una sezione non esistente del documento.

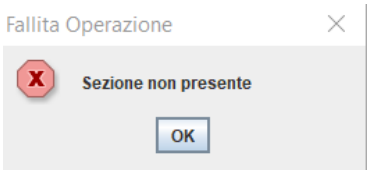

Figura 33 – Messaggio di errore "Sezione non presente"

- Modifica di un documento non esistente o di cui non si è collaboratori.

Figura 34 – Messaggio di errore "Modifica fallita"

- Inserimento di lettere nel campo dove è richiesto il numero di sezione da modificare.

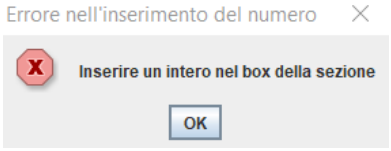

Figura 35 – Messaggio di errore "Numero non riconosciuto"

4.2.6 Logout dell'utente

Il pulsante permette di scollegare l'utente dal servizio TURING, liberando le risorse dell'interfaccia corrente e tornando all'interfaccia di "Registrazione e Login". Aggiungendo un **MouseListener** al pulsante del logout è stato possibile fargli cambiare **Image** ogni qual volta l'utente lo attraversa con il mouse .

Figura 36 – Bottone del logout al passaggio del Mouse

Inoltre, è stato gestito il corretto logout dell'utente anche se viene forzato il logout premendo la "X" di chiusura dell'interfaccia.

4.3 Interfaccia di Editing (EditingGUI.java)

Dopo aver richiesto la modifica di una sezione di un documento, il sistema aprirà una nuova interfaccia in cui sarà possibile messaggiare con tutti gli altri utenti che stanno modificando il documento. Come nell'interfaccia precedente è stato inserito il nome dell'utente padrone della sessione, in modo da poter gestire agilmente più client.
La chat è implementata mediante l'uso di Multicast UDP. Un utente, come già detto, entra a far parte del gruppo del documento al momento dell'avvenuta richiesta di editing, e smette di farne parte non appena esce dall'interfaccia caricando sul sistema il documento modificato. Questo permette di limitare l'invio dei messaggi ai soli utenti interessati, in un dato momento, alla modifica del documento.

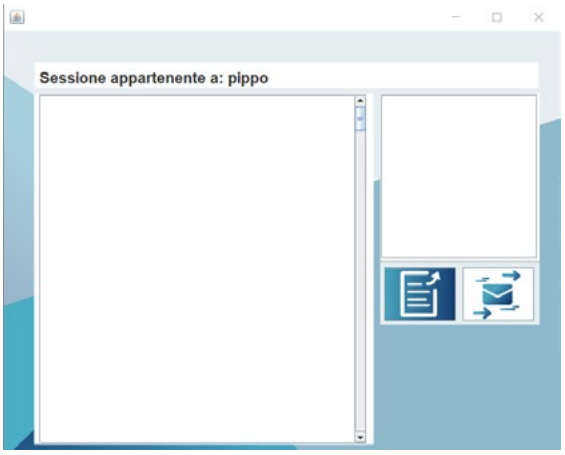

Figura 37 – Interfaccia di "Editing"

4.3.1 Inviare un messaggio in chat

Il pulsante in basso a destra, dell'area di inserimento dei messaggi, permette ad un utente di inviare un messaggio sulla chat. Nel caso in cui il messaggio superi l'apposita area di testo, verrà attivato uno ScrollPane utile allo scopo.

4.3.2 Upload della sezione modificata – fine dell'editing

Il pulsante in basso a sinistra, dell'area di inserimento dei messaggi, permette di terminare la modalità di editing e tornare alla schermata precedente. A quel punto, sempre tramite una gestione NIO, il file verrà ricaricato sul sistema con le modifiche effettuate dall'utente.

4.3.3 Listener della Chat

In fase di editing esiste un Thread denominato **ChatHandler** (ChatHandler.java) che controlla l'esistenza di nuovi messaggi, appendendoli all'area di Chat dell'interfaccia utente che ha creato il suddetto Thread. La gestione dei messaggi della chat viene effettuata tramite protocollo UDP, quindi con l'uso di DatagramPackets. Il Thread listener per la chat viene terminato quando si lascia la fase di editing di una sezione, settando a 0 la guardia del suo ciclo di attesa.

5 OPERAZIONI CLIENT – SERVER

Questo paragrafo si occupa di dare una visione dei messaggi che vengono scambiati tra client e server per gestire le operazioni. Una spiegazione più dettagliata dei messaggi scambiati è presente all'interno del file **Operazioni.txt** contenuto nella cartella del progetto.

Operazione di login di un utente.

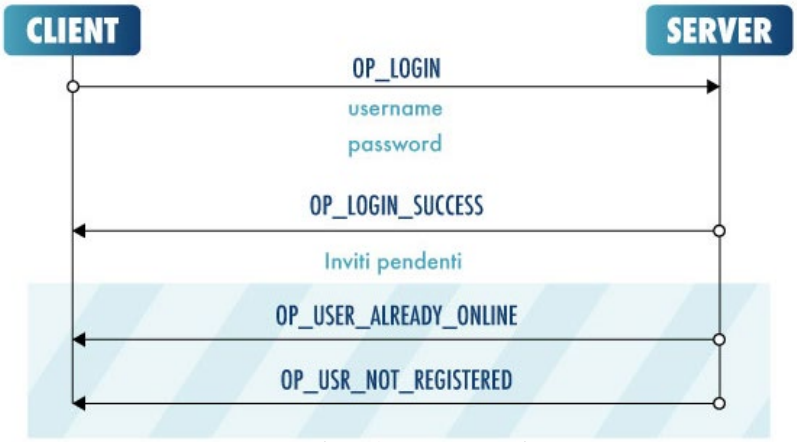

Figura 38 – Flusso di messaggio per il Login

Operazione di creazione di un nuovo documento.

Figura 39 – Flusso di messaggi per la creazione di un Documento

Operazione di Logout di un utente.

Figura 40 – Flusso di messaggi per il logout di un utente

Operazione di Invito alla collaborazione.

Figura 41 – Flusso di messaggi per l'invito di un utente

Operazione di Modifica di una sezione.

Figura 42 – Flusso di messaggi per la modifica di una sezione

Operazione di conclusione della modifica di una sezione.

Figura 43 – Flusso di messaggi per la terminazione dell'edit

Operazione di visualizzazione della lista dei documenti.

Figura 44 – Flusso di messaggi per la visione della lista dei documenti

Operazione di visione di un documento.

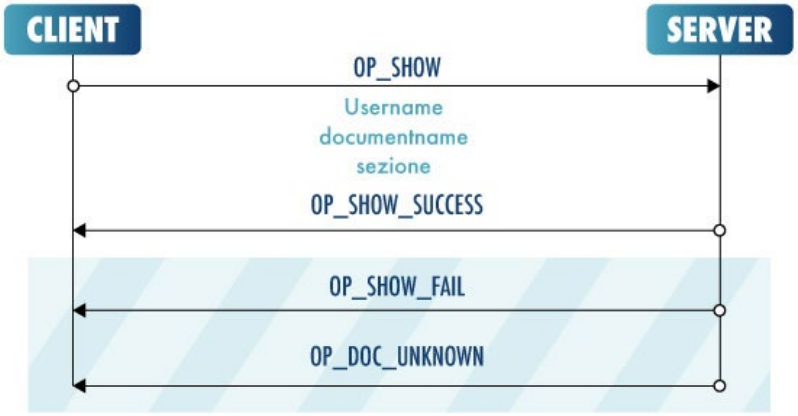

Figura 44 – Flusso di messaggi per la visione della lista dei documenti

6 TESTING

Per sviluppare e testare il progetto è stato utilizzato l'ambiente di sviluppo integrato Eclipse versione 4.9.0. Si ricorda che se si usano altri ambienti si potrebbero avere dei problemi di condifica nelle parole con accento, presenti nelle numerose notifiche pop-up del servizio.

Per eseguire il progetto bisogna attivare prima il server presente in **TuringServer.java**. A server attivo, eseguire uno o più client presenti in **TuringClient.java** Durante lo sviluppo del progetto sono stati effettuati i seguenti test:

1. Esecuzione di un server e più client
    - 1.1 Esecuzione di server e client con configurazioni di default.
    - 1.2 Esecuzione di server e client con configurazioni prese dal file di configurazione presente in **src/DATA/turingServer.conf** (scelta consigliata per avere coerenza tra le configurazioni del client e del server).
2. Registrazione un client
    - 2.1 Registrazione di un client non registrato.
    - 2.2 Registrazione di un client già registrato.
    - 2.3 Registrazione di un client inserendo in almeno uno dei due campi richiesti ("Nome utente" e "Password") un numero di caratteri superiore al numero massimo previsto dalle configurazioni (nei test effettuati questo numero è uguale a 30 caratteri).

3. Login di un client

- 3.1 Login di un client offline, che non ha ricevuto inviti a collaborare mentre era offline.
- 3.2 Login di un client offline, che ha ricevuto inviti a collaborare mentre era offline.
- 3.3 Login di un client già online.
- 3.4 Login di un client non registrato.
- 3.5 Login di un client, logout e login successivo.

4 Creazione di un nuovo documento

- 4.1 Creazione di un documento, con numero di sezioni che va da 2 al numero massimo consentito (compresi).
- 4.2 Creazione di un documento già esistente.
- 4.3 Creazione di un documento con numero di sezioni minore di 2 o maggiore del numero di sezioni massimo.
- 4.4 Creazione di un documento con delle lettere presenti nel numero di sezioni richiesto.

5 Invito di un utente alla collaborazione

- 5.1 Invito di un utente (registrato) a collaborare alla modifica di un documento (esistente), mentre l'utente era offline.
- 5.2 Invito di un utente (registrato) a collaborare alla modifica di un documento (esistente), mentre l'utente era online.
- 5.3 Invito di un utente non registrato.
- 5.4 Invito di un utente registrato ad un documento inesistente.
- 5.5 Invito di un utente registrato ad un documento di cui non sei il creatore.
- 5.6 Invito di te stesso.
- 5.7 Invito di un utente già collaboratore.

6 Visione di un documento

- 6.1 Visione di un documento (esistente) di cui si è collaboratore (inserendo una sezione esistente).
- 6.2 Visione di un documento (non esistente).
- 6.3 Visione di un documento di cui non si è collaboratori oppure visione di una sezione non esistente del documento.
- 6.4 Visione del documento completo (inserendo quindi il numero massimo di sezioni possibili, specificato nelle configurazioni).
- 6.5 Visione di una sezione del documento, che qualcuno sta modificando.
- 6.6 Visione del documento intero, mentre qualcuno sta modificando una o più sezioni.

7 Modifica di una sezione

- 7.1 Modifica della sezione di un documento (esistente) di cui sei collaboratore (con nessun altro utente che sta modificando il documento).
- 7.2 Modifica della sezione di un documento (esistente) di cui sei collaboratore (con altri utenti che stanno modificando il documento), e prova di invio di messaggi.
- 7.3 Modifica di una sezione non presente del documento.
- 7.4 Modifica di un documento inesistente.
- 7.5 Modifica di un documento di cui non si è collaboratore.
- 7.6 Modifica di un documento inserendo lettere nel campo dove si richiede il numero di sezioni.
- 7.7 Modifica di più sezioni di documenti diversi contemporaneamente (da utenti diversi).
- 7.8 Modifica di un documento inserendo un numero di sezioni minore di 0 e maggiore del numero di sezioni possibili

8 Upload di una sezione

8.1 Upload della sezione modificata (con conseguente scollegamento dalla chat degli utenti che hanno effettuato l'upload)

9 Visione della lista dei documenti

10 Logout

10.1 Logout di un utente online tramite apposito bottone

10.2 Logout di un utente online tramite "x" in alto a destra (ipotesi di logout forzato)