TURING

disTribUted collaboRative edItiNG

Reti di Calcolatori - Laboratorio

Federico Gerardi
Matricola: 508082
federicogerardi94@gmail.com

May 3, 2019

Abstract

TURING - *disTribUted collaboRative edItiNG* è una piattaforma client-server realizzata come progetto finale per il modulo di Laboratorio dell'esame di Reti di Calcolatori della Laurea Triennale in informatica dell'Università di Pisa. Il progetto si basa sulla creazione di un sistema di document editing multiutente distribuito (simile a quello offerto da Docs di Google), che gestisce permessi di modifica e operazioni di aggiornamento dei contenuti dei documenti esistenti in maniera concorrente e consistente. Questo paper fornirà una panoramica della sua infrastruttura e illustrerà alcune scelte implementative.

Contents

1 Funzioni della piattaforma

Le funzionalità che la piattaforma implementa sono le seguenti:

- Creazione di un nuovo utente;
- Login dell'utente all'interno della piattaforma;
- Inizio della fase di modifica di una specifica sezione di un documento;
- Terminazione della fase di modifica e aggiornamento della relativa sezione sul server;
- Visualizzazione di una sezione del documento;
- Visualizzazione di un intero documento
- Operazioni per l'invio/ricezione di messaggi in chat condivisa tra gli editor di più sezioni appartenenti allo stesso documento;
- Condivisione dei permessi di accesso ad un documento di cui si è i proprietari ad altri utenti;
- Gestione delle notifiche generate in seguito alla ricezione dei permessi di accesso ad un documento.

2 Compilazione ed Esecuzione

Compilazione La compilazione viene gestita attraverso il toolkit *Maven*, che ci permette di definire delle apposite routine[1] che si occuperanno di effettuare il packing sia del client che del server di TURING in formato JAR.

Listing 1: "Compilazione tramite Maven"

```
$ mvn package
...
[INFO]
------------------------------------------------------------------------

[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------

[INFO] Total time:  14.655 s
[INFO] Finished at: 2019-xx-xxTxx:xx:xx+xx:xx
[INFO]
------------------------------------------------------------------------
```

[1] Oltre alle routine di compilazione e packaging, è definita anche una routine per *pulire* attraverso il comando "*mvn clean*"

A seguito della compilazione, dunque, verranno generati i seguenti file:

- ./target/TURING-Client.jar
- ./target/TURING-Server.jar

Esecuzione Per eseguire i due JAR basta utilizzare il comando *java -jar*, passando,come argomento, il file da eseguire.

Listing 2: "Esempio di esecuzione di TURING-Server.jar

```
$ java -jar ./target/TURING-Server.jar -h
```

3 Interfaccia utente

3.1 Argomenti a Linea di Comando

Utilizzando alcuni argomenti da linea di comando, è possibile specificare alcune preferenze[2] del comportamento sia del client che del server.
In particolare, questi sono i parametri di connessione personalizzabili:

- *–tcp-command-port*: Numero di porta utilizzato per la connessione relativa allo scambio di comando/responso;
- *–udp-multicast-port*: Numero di porta utilizzato[3] per lo scambio di messaggi multicast;
- *–rmi-port*: Numero di porta utilizzato per la connessione TCP sfruttata per effettuare chiamate RMI[4];
- *-data-dir*: Path utilizzato per effettuare la memorizzazione dei dati[5];
- *–server-address*: Indirizzo IPv4 del server[6];
- *–config-file*: Nome del file JSON di configurazione;

[2] È possibile avere la lista completa attraverso l'invocazione dei due programmi con il flag *-h* o *–help*

[3] Opzione disponibile unicamente sul client

[4] L'unica funzione che sfrutta RMI è la registrazione di nuovi utenti

[5] Viene utilizzato dal client per memorizzare i file locali delle sezioni in modifica e dal server per memorizzare la serializzazione degli oggetti utili al mantenimento dei dati relativi agli utenti e ai documenti

[6] Opzione disponibile unicamente sul client

File JSON di Configurazione Per non dover utilizzare molti argomenti da riga di comando in ambienti in cui sorge la necessità di utilizzare molti parametri i cui valori differiscono da quelli di default, TURING mette a disposizione[7] la possibilità di utilizzare un file JSON di configurazione da passare come unico argomento a riga di comando durante l'esecuzione dell'applicazione.
Le stringhe utilizzabili all'interno dell'oggetto JSON principale sono le seguenti[8]:

- *TCP_PORT*
- *UDP_PORT*
- *RMI_PORT*
- *DATA_DIR*
- *SERVER_ADDRESS*

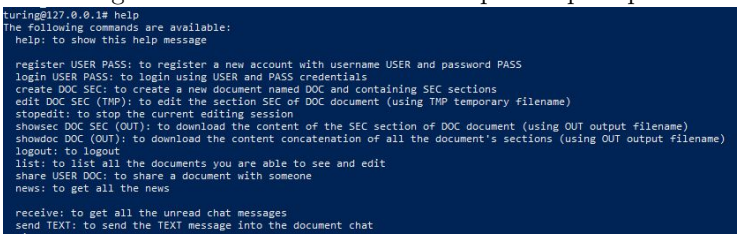
Listing 3: "JSON File - Esempio"

```
{
        "TCP_PORT": 9658,
        "DATA_DIR": "/home/user/TURING/",
        "RMI_PORT": 15698
}
```

3.2 CLI

È possibile interagire con il sistema attraverso l'apposito client. Questo fornisce un interfaccia interattiva a riga di comando, con la quale è possibile interagire grazie all'inserimento iterativo di comandi utilizzando il relativo prompt.

Figure 1: TURING Client - Esempio del prompt



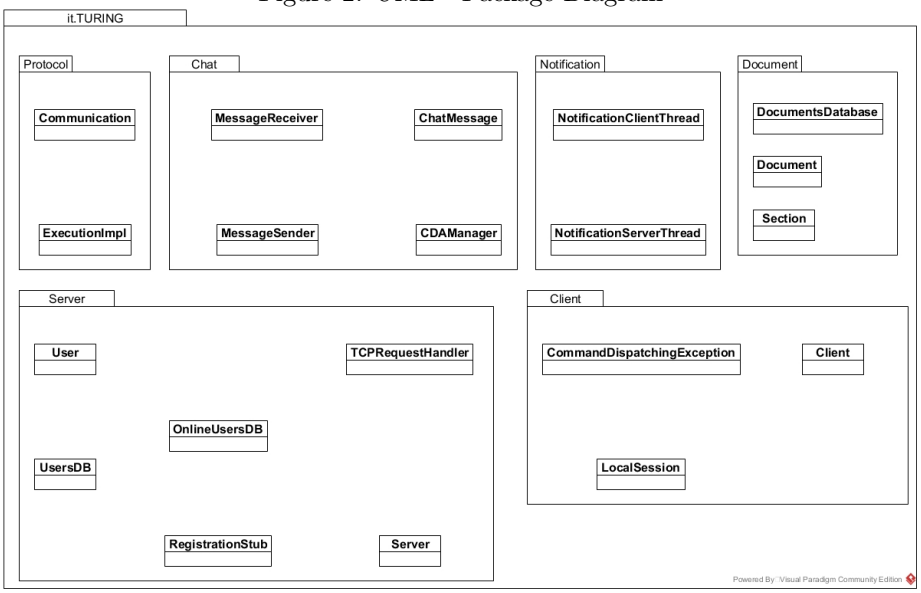
[7] Sia il client, che il server mettono a disposizione questa funzionalità
[8] Corrispondono a quelle illustrate precedentemente come argomenti a linea di comando

4 Struttura dei Package

I package principali, figli del root package *it.azraelsec*, che fanno parte del progetto sono i seguenti:

- `Chat` - Cassi che interessate nella gestione del servizio di messaggistica multicast UDP
- `Client` - Classi costituenti il client del sistema TURING
- `Document` - Classi relativi alla rappresentazione dei documenti e delle sezioni costituenti[9]
- `Notification` - Classi per la gestione delle notifiche lato client e per la loro generazione lato server
- `Protocol` - Classi ed interfacce atte alla gestione del protocollo di rete *low-level*
- `Server` - Classi costituenti il server del sistema TURING, relative alla gestione del concetto di *Utente* ed al ciclo di vita delle sue sessioni

Figure 2: UML - Package Diagram



[9] I Documenti, infatti, sono formati in realtà da un'aggreazione di differenti Sezioni

5 Server

Il server ha una struttura multi-thread: ogni nuova connessione viene gestita da un differente *TCPRequestHandler*[10], il cui ciclo di vita viene incapsulato all'interno di un *ThreadPoolExecutor*[11].

Listing 4: "Gestione di una nuova connessione"

```
while(true) {
        Socket socket = TCPServer.accept();
        System.out.println("New TCP connection: " + socket.
            getRemoteSocketAddress().toString());
        TCPConnectionDispatcher.submit(new TCPRequestHandler(
            onlineUsersDB, usersDB, documentDatabase, cdaManager,
            socket));
}
```

L'oggetto Server, attraverso il metodo *bootstrap*, effettua l'inizializzazione di tutti gli oggetti interni necessari all'esecuzione del ciclo di vita dell'applicazione e tutte le proprietà, considerando le impostazioni scelte dall'utente in fase di avvio del processo[12]:

- Inizializzazione dei valori delle configurazioni;
- Controllo della directory di lavoro del server[13];
- Carica le informazioni del database degli utenti[14];
- Carica le informazioni del database dei documenti[15];
- Configura lo stub RMI per la chiamata *register* remota;
- Registra un handler per gestire la chiusura del processo.

Persistenza delle informazioni Come detto procedentemente, il server inizializza il database dei documenti e quello degli utenti effettuando una ricerca all'interno della cartella di lavoro per trovare i file **db.dat** e **docs.dat**. Se questi vengono individuati, Server tenta di effettuare una deserializzazione per ricostruire gli oggetti originali, altrimenti vengono utilizzati degli oggetti vergini non contenenti alcun dato.

La serializzazione, invece, avviene all'interno della routine associata all'hook di terminazione, che viene richiamata in fase di chiusura del processo.

[10] Che eredita da *Runnable*

[11] Si è scelto di utilizzare un ThreadPoolExecutor di tipo *CachedThreadPool*

[12] L'ordine di priorità delle impostazioni è il seguente: riga di comando $\succ$ file di configurazione $\succ$ valori di default

[13] Controlla se la directory esiste ed è valida, altrimenti la crea

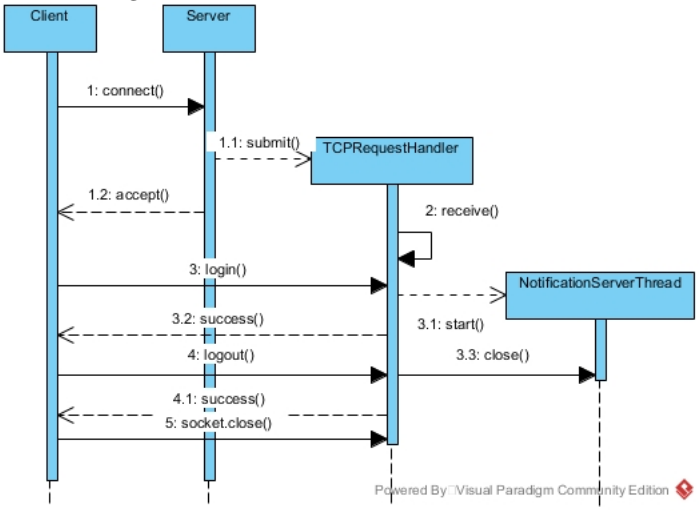
[14] Astratto dalla classe *UsersDB*

[15] Astratto dalla classe *DocumentsDatabase*

Sessione Si è deciso di gestire lo scambio di messaggi tra client e server mantenendo attiva una connessione TCP persistente, ma sfruttando comunque un meccanismo di identificazione univoco della sessione [16] attraverso l'impiego un token, che viene generato dal Server in seguito alla richiesta di login [17] e distrutto in seguito a quella di logout. La motivazione che sta dietro a questa scelta progettuale è quella di porre alla base del meccanismo di identificazione una struttura facilmente ampliabile: in questo modo, è semplice implementare un meccanismo che permetta di impiantare direttamente il token all'apertura del client e di permettere così ad altri di utilizzare il proprio "profilo" senza la necessità di fornirne le credenziali in chiaro.

Thread Life-cycle Alla richiesta di connessione da parte del client sul socket principale, il server smista il canale sul socket secondario e passa il suo riferimento al *TCPRequestHandler*, il quale si occuperà di effettuare il dispatching dei comandi e richiamerà gli opportuni handler. Alla richiesta di **login**, il server creerà una nuova istanza del *NotificationServerThread*[18], che rimarrà attivo per tutto il tempo della durata della sessione.

Figure 3: Server - Ciclo di vita dei Thread



[16] Per *sessione* s'intende tutta la sequenza di comandi TURING che intercorrono tra una richiesta di login e la rispettiva richiesta di logout

[17] Il token è una stringa a 256-bit formata dallo SHA-256 della concatenazione del timestamp della richiesta e dell'hash della password dell'utente

[18] Questo, a differenza di quanto si potrebbe immaginare, in realtà funge da client per una "reverse connection" che viene instaurata allo specifico scopo di gestire richieste e risposte relative al pulling di notifiche

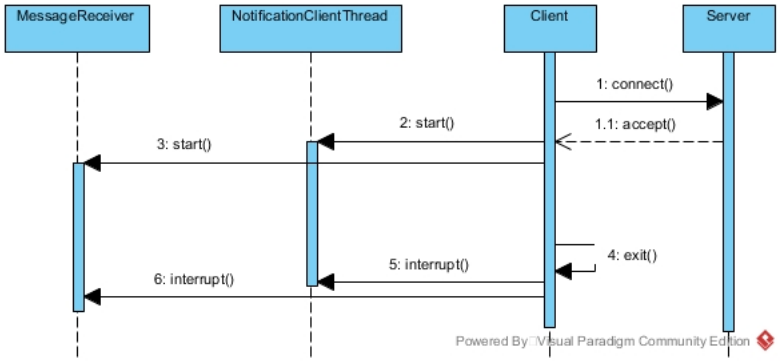
6 Client

Il programma client tenta di stabilire immediatamente una connessione TCP con il server: in caso di esito positivo, dà la possibilità all'utente di interagire con il sistema TURING, e interrompe con esito negativo l'esecuzione in caso contrario.

Il client non si serve di alcuna persistenza dei dati, ma necessita in ogni caso di una directory di lavoro da sfruttare per la memorizzazione dei file che gli vengono inviati dal server per la modifica delle sezioni o per la ricezione del contenuto attuale dei documenti. Il meccanismo utilizzato per la verifica dell'esistenza e validità di tale directory è identico a quello utilizzato dal server.

Thread Life-cycle Lo scambio di comandi e risultati avviene all'interno del thread principale, ma viene delegata a thread secondari la gestione delle notifiche e la ricezione dei messaggi multicast diretti al gruppo di cui il client è membro.

Figure 4: Client - Ciclo di vita dei Thread



7 Protocollo di scambio

Il protocollo utilizzato per lo scambio di messaggi è molto semplice e si basa sulla ricezione da parte del server di comandi e del successivo invio di esito dell'operazione al client. Ogni operazione, infatti, può portare all'invio da parte del server di una risposta **success** o **failure** [19].

Dunque, tutti i tipi di messaggi scambiati sono inglobati all'interno dell'enum **Commands**:

- LOGIN
- LOGOUT
- CREATE
- EDIT
- EDIT_END
- SHOW_SECTION
- SHOW_DOCUMENT
- LIST
- SHARE
- SUCCESS
- FAILURE
- NEW_NOTIFICATIONS
- EXIT

Per ognuno di questi, viene definito un array di tipi, così da dare la possibilità al client di controllare che il tipo di dato degli argomenti inviati sia corretto e permettere al server di inferire l'ordine ed il tipo di *receive()* da effettuare per ricostruire i dati inviati. Nella sezione 7.1 verrà chiarito meglio in che modo questi vengano scambiati a partire dai loro tipi.

Il protocollo prevede anche delle procedure per lo scambio di stream di dati, utili per l'invio e ricezione da parte del server del contenuto dei file rappresentanti le sezioni dei documenti o la loro concatenazione[20] . L'invio di stream dev'essere necessariamente preceduto da un normale invio di esito da parte del server per evitare la rottura del protocollo.

Approccio Funzionale L'approccio utilizzato nella stesura del codice relativo all'implementazione del protocollo è fortemente funzionale e segue le tecniche più moderne, coinvolgendo la definizione di appositi handler chiamati in base all'esito delle operazioni. Il listing 5 è un chiaro esempio di come la programmazione funzionale venga impiegata nell'utilizzo del protocollo.

7.1 Rappresentazione dei dati

Vengono utilizzati degli oggetti di tipo **DataInputStream** e **DataOutputStream** rispettivamente per la ricezione e l'invio dei dati sul socket, per rius-

[19] Si è scelto di generalizzare quanto più possibile il concetto di *messaggio*, così da includervi anche quelli di risposta

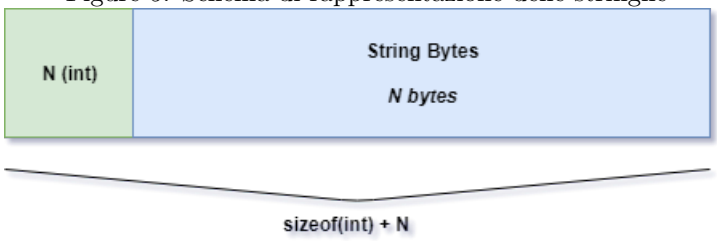
[20] Infatti, durante l'esecuzione del comando **SHOW_DOCUMENT**, il server si serve della concatenazione di più stream di file per crearne uno unico da poter inviare al client, sfruttando la funzione appena descritta

Listing 5: "Frammento in cui si evidenzia l'approccio funzionale del protocollo"

```
private void documentsList() {
        if (session != null)
                Communication.send(
                        clientOutputStream,
                        clientInputStream,
                        System.out::println, System.err::println,
                        Commands.LIST);
        else System.err.println("You're not logged in");
}
```

cire ad inviarne la corretta rappresentazione in byte[21]. Se per gli interi questo approccio risulta sufficiente ad assicurarne correttamente lo scambio, per le stringhe si è dovuto far affidamento all'invio di una informazione addizionale: il numero di byte che le compongono. Infatti, ogni volta che si presenta la necessità di inviare una stringa, si procede prima alla *send()* del numero di byte che la compongono e successivamente a quella dell'array dei dati. In questo modo, utilizzando l'approccio contrario, è possibile sapere di preciso quanti byte appartenenti alla stringa da ricevere prelevare dal buffer.

Figure 5: Schema di rappresentazione delle stringhe



7.2 Sistema di notifiche

Oltre al sistema di interpretazione ed esecuzione dei comandi, anche il sistema di notifiche viene gestito attraverso un'implementazione dello stesso protocollo client-server. Infatti, quando l'utente effettua il login tramite il client, viene stabilita una seconda connessione TCP dal server verso il client.
È da notare che:

1. il client TURING è in questo scenario il server[22] delle notifiche (la connessione è, come detto precedentemente, instaurata al contrario);

[21] Molto utile per l'invio corretto di interi, per esempio, evitando di dover effettuare interpretazione di testo contenente la loro rappresentazione

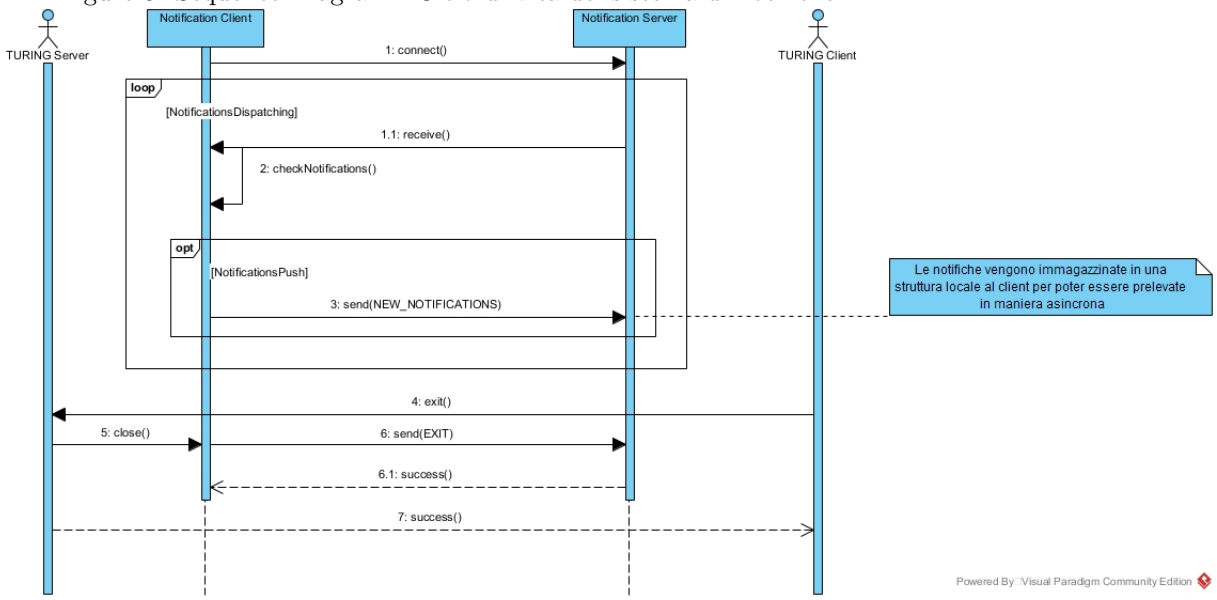
[22] Il server delle notifiche adotta **multiplexing** per gestire in maniera corretta segnali del processo

2. il server delle notifiche controlla in maniera periodica se ci sono notifiche nuove da inviare al client e, in questo caso, effettua una *send()* utilizzando i metodi definiti dal protocollo di scambio dei messaggi;

3. il client delle notifiche rimane in ascolto per eventuali notifiche inviate dal server, registrando opportuni handler per la gestione di questa casistica;

4. quando il client riceve un comando di chiusura dal prompt, comunica la cosa al server attraverso l'impiego della connessione di controllo principale. Sarà poi il server, attraverso il proprio **NotificationServerThread**, a comunicare al **NotificationClientThread** la chiusura della connessione per le notifiche.

Figure 6: Sequence Diagram - Ciclo di vita del sistema di notifiche



8 Documenti

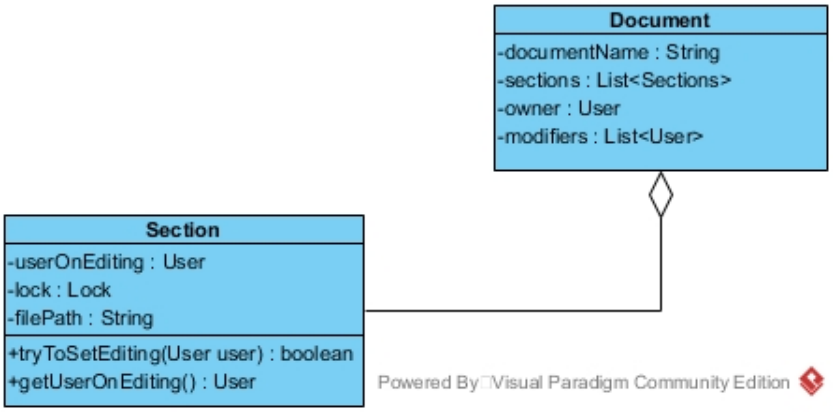
Tutti i documenti gestiti dalla piattaforma TURING sono astratti dalla classe **Document**, che mantiene i riferimenti alle singole sezioni, a loro volta rappresentate dalle istanze della classe **Section**. Come illustrato dal class diagram 7, il documento si occupa di mantenere i riferimenti agli utenti aventi i permessi di accesso al contenuto delle singole sezioni[23].

[23] Vengono mantenuti i riferimenti sia al profilo del creatore del documento, sia alla lista di coloro i quali hanno accesso alla struttura come *modifiers*. In questo modo, il sistema è in grado di negare la possibilità a questi ultimi di aggiungere altri utenti alla lista degli aventi accesso, lasciando questa possibilità unicamente ai creatori, come da specifiche

Tutte le operazioni su filesystem relative alla gestione dei documenti sono effettuate tramite NIO.

Accesso Esclusivo La gestione dell'accesso esclusivo al contenuto delle sezioni viene lasciata agli oggetti **Section**, che mantengono tra le proprietà d'istanza una **Lock**[24], la quale viene utilizzata per l'accesso alle sezioni critiche relative all'assegnazione dei riferimenti dell'utente che è in *Editing Mode*. Per comprendere, infatti, se esiste già o meno un utente in questa fase sulla sezione obiettivo, si analizza la variabile **userOnEditing** e, se questa è *null*, allora si procede all'assegnazione del riferimento dell'utente richiedente.

Figure 7: Schema di rappresentazione delle stringhe



Memorizzazione Locale A livello fisico ogni documento è una vera e propria directory, mentre le sezioni sono file al suo interno. Al momento della creazione di un nuovo documento, vengono generate le sezioni aventi un filename randomico[25]. Sarà poi l'oggetto **DocumentsDatabase** ad occuparsi di mantenere i riferimenti alla lista completa dei documenti[26] e ad interfacciarsi con il resto della piattaforma.

9 Chat

Ogni peer della chat è un client/server multicast che si lega ad un *multicast group* di riferimento per i gruppi di utenti che stanno editando sezioni appartenenti ad

[24] Si è scelto di utilizzare un oggetto di tipo **ReentrantLock**.

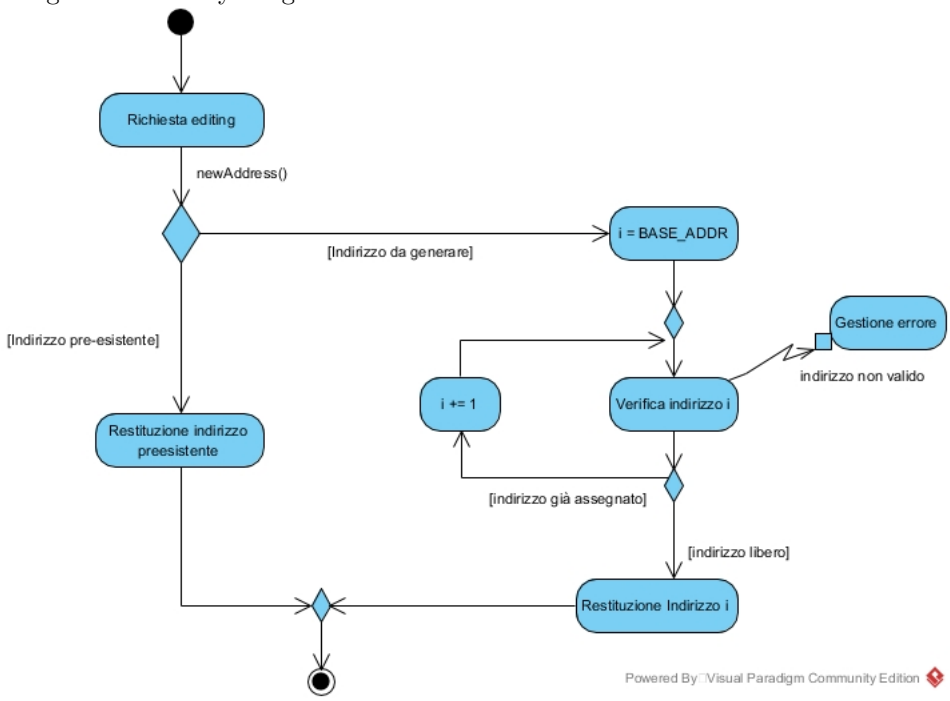
[25] Si è scelto di utilizzare il valore del timestamp misurato al momento della creazione della prima sezione e successivamente incrementato di uno per ognuna delle restanti

[26] In realtà, si sfrutta una struttura *factory* per la generazione dei documenti: non viene mai creato un **Document** utilizzando il suo costruttore, ma si sfrutta il metodo **createNewDocument** del **DocumentsDatabase**, che a cascata provocherà la generazione di documento e sezioni.

uno stesso documento: se non esistono su di esso, la relativa chat non dovrebbe concettualmente esistere.

Assegnazione Dinamica dei Gruppi Multicast Per implementare questo comportamento, ho scelto di generare e gestire gli indirizzi delle chat in maniera dinamica, attraverso un algoritmo di generazione molto semplice illustrato in figura 8 . A gestire l'allocazione degli indirizzi è l'oggetto **CDAManager**[27]. Questo si serve della rappresentazione intera decimale degli indirizzi IPv4 per maneggiarli in maniera più semplice e gestirne meglio l'incremento.

Figure 8: Activity Diagram - Generazione dinamica indirizzo di multicast



Quando tutti gli editor terminano la modifica della sezione, l'indirizzo diventa nuovamente disponibile e riutilizzabile da un altro gruppo multicast.

Comunicazione tra i Client Il server TURING non gioca alcun ruolo nella comunicazione tramite il servizio di chat se non quello di fornire, come descritto in precedenza, l'indirizzo di multicast relativo al canale del documento che si sta editando. Questa struttura ha permesso di snellire il processo di funzionamento del servizio, evitando di interporre inutilmente il server in funzione di *proxy* delle richieste tra i fruitori.

[27] CDAManager := Chat Dynamic Address Manager

I messaggi vengono ricevuti dall'oggetto **MessageReceiver** e inviati tramite il **MessageSender**, incapsulando le informazioni all'interno di **ChatMessage**. I messaggi ricevuti vengono immagazzinati all'interno di una struttura interna al receiver, che li mantiene fino a quando non ne viene richiesto il contenuto in maniera asincrona attraverso il comando *receive*. A quel punto, i messaggi vengono riordinati temporalmente[28] e stampati a schermo. Una volta stampati, i messaggi vengono persi e non mantenuti in memoria da alcun nodo.

10 Librerie Addizionali

Sono stati utilizzate le seguenti librerie e tool esterni:

- Apache Maven
- Argparse4j
- JSON

[28] Viene sfruttato il campo **time** e l'interfaccia **Comparable** della classe **ChatMessage** per ripristinare l'ordine originale di invio dei messaggi in caso di ritardo di ricezione.