TURING

disTribUted collaboRative edItiNG

Reti di Calcolatori - Laboratorio

Federico Gerardi
Matricola: 508082
federicogerardi94@gmail.com

April 30, 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] 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, le seguenti sono i parametri di connessione personalizzabili attraverso gli argomenti a riga di comando:

- *–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*

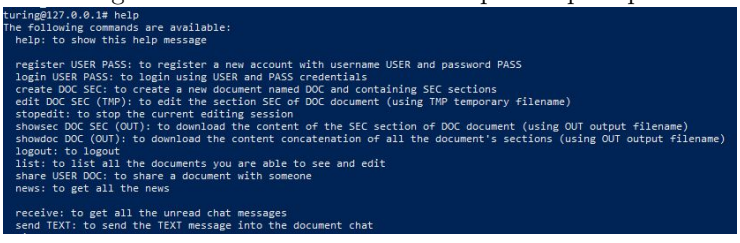
Listing 3: "JSON File - Esempio"

```
{
        "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



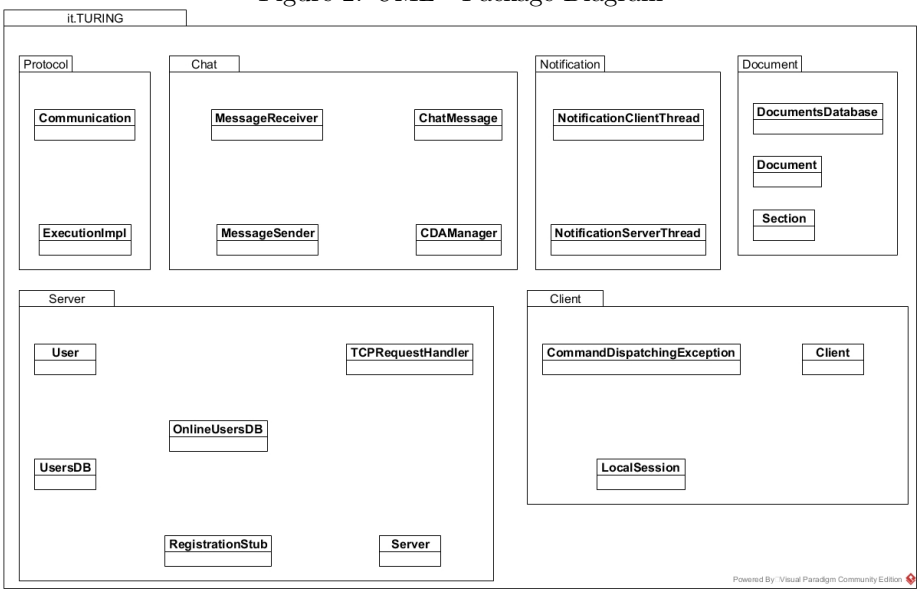
[7] Sia il client, che il server mettono a disposizione questa funzionalità
[8] Corrispondono a quelle illustrate precedentemente come argomenti a linea di comando

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 degli oggetti, invece, avviene all'interno della routine associata all'hook di terminazione, che viene richiamata in fase di terminazione 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

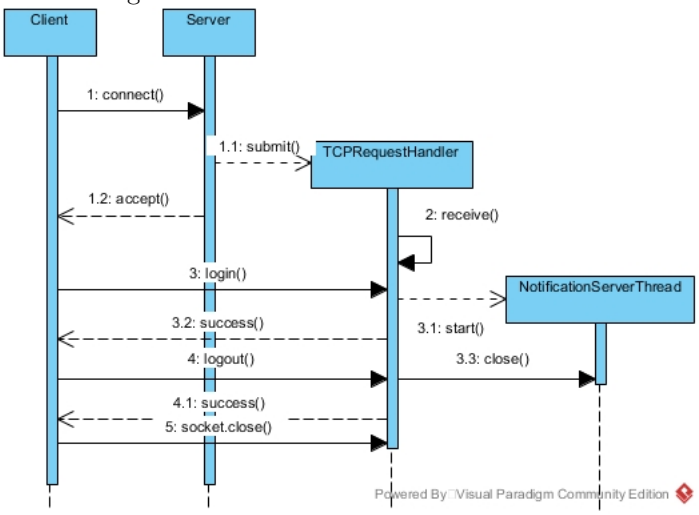
[14] Astratto dalla classe *UsersDB*

[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 implementativa è quella di porre alla base del meccanismo di identificazione una struttura quanto più semplice da modificare e da ampliare: così strutturata, è possibile lasciare la possibilità di implementare un meccanismo che permetta di impiantare direttamente il token all'apertura del client e di poter così permettere ad altri di utilizzare il proprio "profilo" senza la necessità di fornire le proprie 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

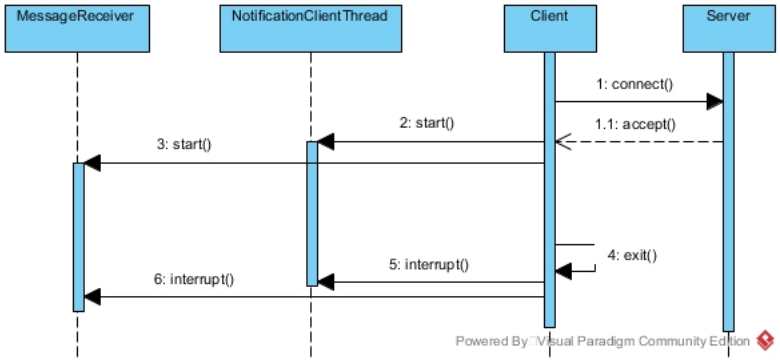
6 Client

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 attraverso una riga di comando, e interrompe con esito negativo l'esecuzione in caso contrario.

A differenza del server, 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 permettere 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