Documentazione TURING

(disTribUted collaboRative edItiNG - Stefano Spadola 534919)

Indice generale

Architettura e scelte progettuali.

Descrizione ad alto livello dell’architettura.

Il progetto è stato curato seguendo le specifiche dalla consegna, ovvero si è scelto di implementare un **Server multithread**, si attiva quindi un thread per connessione, in modo tale da servire più client in maniera parallela.

Il protocollo di comunicazione tra **Client** e **Server** è effettuato tramite lo scambio di messaggi sincroni via **TCP**, ogni client infatti invia dei messaggi in formato testuale aspettandosi una relativa risposta che potrà avere esito positivo (cioè il messaggio di risposta richiesto come ad es. la visualizzazione di un documento) o esito negativo (cioè un codice di errore, ad es. “Documento non esistente”).

Inoltre viene implementato un semplice protocollo di comunicazione tra client, che in uno stesso istante editano uno stesso documento, ovvero una sorta di chat P2P che viaggia su connessioni **UDP** di tipo **multicast**.

Strutture dati utilizzate.

Come precedentemente detto, si è scelto di implementare TURING in versione multithread, questo per garantire una gestione quanto più parallela degli utenti. Un thread per utente viene creato durante la fase di connessione vera e propria ed aggiunta ad un **newFixedThreadPool()**, impostato staticamente di dimensione 100, in quanto a scopo didattico non si è posto l’obiettivo di una ricerca del valore ottimale. In un contesto reale risulta chiaro che debba essere configurato in base alle esigenze sia della

macchina che ospita il Server e sia dell'utenza media di TURING. Un'altra scelta poteva essere quella di utilizzare un newCachedThreadPool() impostando a codice delle soglie da mantenere, oppure come da consegna utilizzare i SocketChannel ed effettuare il multiplexing dei canali.

Le strutture dati utilizzate per la memorizzazione degli utenti e le informazioni relative ai file editati sono racchiuse rispettivamente nelle classi **UsersDB** e **FilesDB** che immagazzinano dati in maniera efficiente e concorrente per mezzo di due **ConcurrentHashMap<Chiave,Valore>**. Attraverso questa struttura fornita da JCF, siamo in grado di fare inserimenti multipli con un alta scalabilità, in quanto la sua implementazione è strutturata in modo tale da bloccare in maniera adattiva la tabella per sezioni, il che la rende molto favorevole per applicazioni che vi accedono in maniera concorrente.

La chiave utilizzata per mappare gli utenti nella tabella Hash in UsersDB non è altro che l'**username** dell'utente che si registra, in quanto dalle specifiche del progetto deve essere univoco, mentre per i file in FilesDB è il **fileID**, ovvero la *concatenazione tra nome file e username*, che rende la chiave univoca (altre scelte sarebbero potute ricadere su un generatore di chiavi randomico).

//es. Username: Stefano, vuole creare File: "documento1" → *fileID="documento1Stefano"*

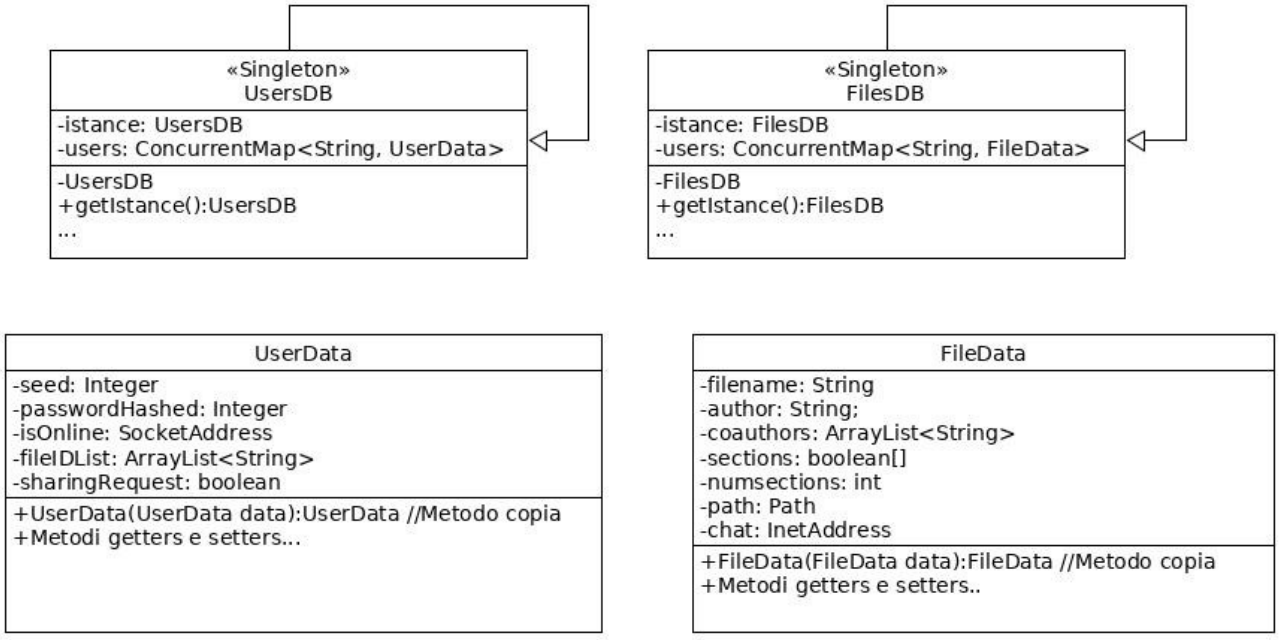


Figura 1: Diagrammi delle classi delle strutture dati utilizzate da TURING

UserData e **FileData** contengono le informazioni/metadati rispettivamente per utente e per file, come ad esempio nel caso degli utenti la password (in formato hash), lo stato (se online o meno), la lista dei fileID che può editare/visionare, e eventuali richieste di condivisione file. Mentre nel caso dei file, FileData immagazina informazioni quali il nome del file, l'autore, il numero delle sezioni, eventuali

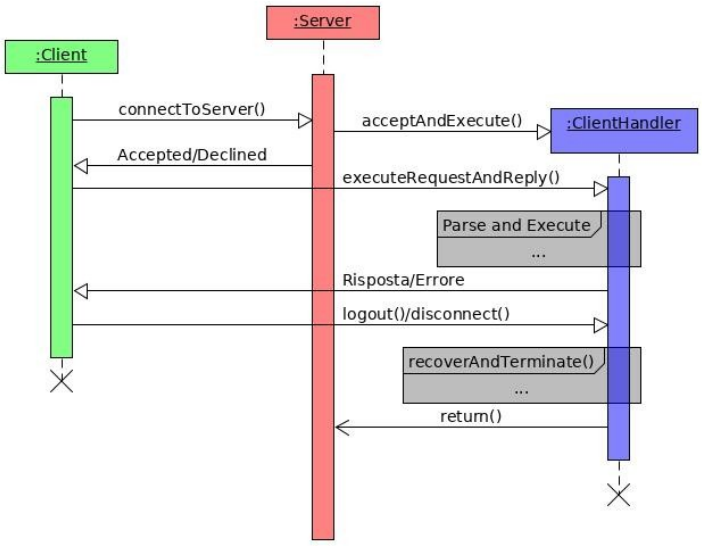
coautori e l’indirizzo multicast che gli utenti, che modificano un determinato file, utilizzeranno per chattare.

Struttura del Client.

Il Client invece si presente principalmente monothread, se non per la gestione della chat, dove nella fase di editing file, viene generato un thread responsabile per la ricezione di messaggi da parte di altri client, che in un dato momento, editano lo stesso file.

Il ciclo di vita è quello che ci si aspetta, ovvero un ciclo che legge ripetutamente da CLI gli input dell’utente, dopodiché procede ad un pre parsing interno al Client stesso, per evitare lo scambio inutile di messaggi formattati male con il Server, e invia successivamente la query al server, aspettandosi una risposta in base alle casistiche.

Thread, gestione concorrenza e sincronizzazione.



La concorrenza è gestita grazie alle proprietà della ConcurrentHashMap che consente accessi Thread Safe senza bloccare interamente la tabella, tutto ciò a scapito della sincronizzazione. Se ciò può sembrare un problema, dato che i vari ClientHandler parallelamente effettuano modifiche e aggiornamenti alle tabelle condivise, in realtà non lo è grazie ad alcune operazioni garantite essere atomiche dalla documentazione Java.

I casi di sincronizzazione vengono così risolti lasciando le strutture dati consistenti nel tempo.

Figura 2: Diagramma di sequenza semplificato.

Le operazioni garantite atomiche e che vengono usate in TURING sono:

- **putIfAbsent**(key,value);
- **replace**(key,oldValue,newValue);

e vengono usate corrispettivamente: in fase di inserimento (*putIfAbsent(...)*) ovvero quando un utente si registra o viene creato un nuovo file, e in fase di modifica (*replace(...)*) ovvero quando si devono eseguire le restati operazioni come ad esempio la condivisione di un documento, o il login di un utente.

La *replace* modifica la entry mappata da una specifica “key” con *newValue* se e solo se vi è mappata l’istanza *oldValue*. Grazie a questo operatore per ogni operazione di modifica delle strutture dati il codice che viene eseguito segue per lo più questa struttura:

1. Viene estratto l’oggetto di chiave “key” (chiamiamolo oldValue).
2. Viene effettuata una Deep Copy del campo “value”, grazie ai metodi **costruttori copia** presenti in FileData e UserData.
3. Si lavora sull’oggetto copiato (chiamiamolo newValue) apportando le modifiche.
4. Si sostituisce l’oggetto con il comando “replace(key,oldValue,newValue)”

Questo metodo ritornerà true se e solo se la sostituzione è andata a buon fine, ovvero se nessun altro Thread ha modificato quella entry in maniera concorrente tra l’estrazione del valore, la sua modifica e il suo rimpiazzo.

In caso contrario, la modifica non avrà effetto, ritornando a chi ne ha fatto richiesta un errore di tipo: CONCURRENCY_ERROR, che verrà trattato in base alle varie casistiche.

Si potrebbe dire che la soluzione attuata per risolvere i problemi di sincronizzazione sia una “*non soluzione*”, in quanto la scelta che viene messa in atto è quella di delegare il chiamante per la gestione dell’errore a favore di una maggiore concorrenza, piuttosto che sincronizzare intere strutture dati che potrebbero ridurre le prestazioni in termini di scalabilità.

Descrizione delle classi.

Server

Contiene il main principale per il Server stesso, prevede 3 fasi in totale:

1. Una fase di *Bootstrap* dove si inizializzano le strutture dati dei file e degli utenti.
2. Il ciclo di vita principale del server, he comprende l’implementazione del servizio di registrazione tramite RMI e successivamente il lancio del servizio che rimane in ascolto su un ServerSocketChannel (implementato da *tcpDeamon()*), ovvero quella parte software responsabile di accettare nuove connessioni via SocketChannel, e di creare un thread dedicato per poi aggiungerlo a un newFixedThreadPool.
3. Una fase di *Shutdown* che è stata lasciata non implementata.

Inoltre vi è contenuto un metodo sincronizzato (*getFreeInetAddress()*) che è responsabile della generazione di indirizzi multicast UPD da assegnare ai file per il servizio di chat. (ATTENZIONE: a scopo didattico il metodo è implementato senza particolari controlli e non prevede il riassegnamento di indirizzi non più utilizzati *Consultare il capitolo: *Problemi, limiti e possibili soluzioni.*).

ClientHandler

È il thread responsabile per la gestione di un singolo utente. Legge via socket un messaggio per volta e risponde sempre via socket con il relativo risultato/errore.

Ha un *Parser* implementato all'interno che controlla la validità delle query lanciate dall'utente ed è responsabile del dispaccio di esse una volta validate. L'handler così processerà la richieste adoperando sulle strutture dati FilesDB e UsersDB.

Sono implementate inoltre le funzioni di *picking*, *upload* e *download* dei file, ovvero funzioni che sono utilizzate dai comandi di <show>,<edit> e <end-edit> e che servono per gestire i file lato Server attraverso le librerie Java NIO (utilizzando i FileChannel).

Inoltre è dotato di una funzione di terminazione (*recoverAndTerminate()*) che si occupa di lasciare le strutture dati in uno stato consistente anche in caso di chiusura improvvisa da parte del client, nonché di terminare il thread chiudendo le risorse utilizzate (Socket, reader e writer).

FilesDB

Come già detto è la struttura responsabile per la memorizzazione dei file e le loro relative informazioni (es autore, numero sezioni...) facendo uso della classe d'appoggio FileData. La struttura cardine è una *ConcurrentHashMap<String,FileData>* dove il campo chiave è rappresentato dal **fileID**, una "concaztenazione tra nome file e nome utente" che rende unica la entry.

//es. Username: Stefano, vuole creare File: "documento1" ==> fileID="documento1Stefano"

La classe è dotata di metodi che servono a manipolare le entry della tabella ed eseguire operazioni di inserzione e/o aggiornamento.

UsersDB

Rappresenta il database che contiene tutti gli utenti del sistema TURING. Anch'esso composto da una ConcurrentHashMap<String,UserData> dove il campo chiave stavolta è rappresentato dall'username degli utilizzatori del sistema, e il campo value invece da UserData che contiene le informazioni relative ad un dato utente.

FileData

È il campo value usato da FilesDB per memorizzare le informazioni relative ai file, comprende i campi: nome file, autore, coautori, il numero delle sezioni, un vettore di booleani che rappresentano le sezioni bloccate in modifica, i percorso dove sono salvati, e un indirizzo multicast udp usato per la chat.

Sono inoltre implementate le relative funzioni per manipolarne i campi.

UserData

È la classe d'appoggio usata da UsersDB per memorizzare le informazioni relative agli utenti quali password in formato hash, la lista dei file editabili/visionabili, l'indirizzo da cui un client può essere collegato o meno e una variabile booleana che indica se vi sono richieste pendenti di condivisione file.

Client

È il main eseguito dagli utenti che vogliono usufruire dei servizi offerti da Turing, offre funzionalità di connessione, disconnessione e registrazione via RMI con il Server.

L'interfaccia è quella di una comunissima CLI che prende in input da un utente una serie di query da far eseguire al server una volta parsate. Implementa infatti un protocollo di "richiesta-risposta" (implementato da *executeRequestReply()*) che invia richieste e attende risposte/errori. Tale protocollo è dotato di un meccanismo built-in che senza l'interazione con l'utente cerca risolvere gli errori di tipo CONCURRENT_ERROR, riprovando a lanciare il comando fino ad un massimo di 5 volte, dopodiché procede a notificare l'utente.

È dotato in oltre di funzioni di *picking*, *downloading* e *uploading* per quanto riguarda la gestione dei file e lo scambio che avviene tra client e server per le operazioni di <show>,<edit> e <end-edit>.

ChatThread

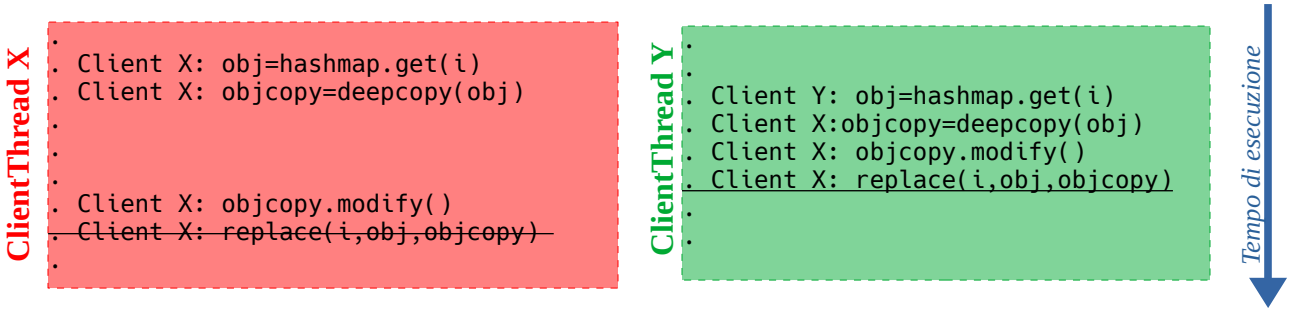
ChatThread è il thread che viene lanciato da un utente (1 per utente) per comunicare con altri utenti che in un dato istante modificano sezioni diverse di uno stesso file. La chat presenta una struttura pressoché P2P, dove l'unica interazione col server avviene solamente mediante lo scambio dell'indirizzo multicast udp durante la fase di edit.

Il thread viene terminato dal Client dopo un messaggio di end-edit.

Problemi, limiti e possibili soluzioni.

Il software così fornito è garantito funzionare nei casi previsti dalla consegna del progetto. Può però presentare qualche bug per la mancata gestione di casi non previsti nonché può portare a stati inconsistenti per via dell'incompleta gestione di tutte le eccezioni meno frequenti.

Uno dei "problemi" che può presentare, seppur il caso sia abbastanza remoto, è quello della **starvation** da parte di uno o più Client. Il motivo riguarda la gestione della concorrenza/sincronizzazione spiegata nel paragrafo "*Thread, gestione concorrenza e sincronizzazione*". Per avere un fenomeno consistente di starvation ogni comando (di modifica) richiesto da uno stesso client X deve essere sempre terminato dopo l'esecuzione del comando impartito da un client Y che modifica la stessa istanza che X ha estratto per processare. Si può, più facilmente che a parole, intuire dal seguente schema:



Risulta però un caso molto improbabile, sia per il grado di asincronia che c'è tra i diversi client sia per la bassa probabilità che sia sempre lo stesso client a non essere servito un numero consecutivo "n" di volte per problemi di concorrenza. Viene così trascurato il problema e delegato al chiamante (client) che cercherà di ri-sottomettere la query al server.

Un'altra soluzione adottabile poteva essere quelle di bloccare le strutture dati con dei monitor (blocchi synchronized) per rendere atomiche le modifiche a scapito però, della concorrenza.

Un altro problema (facilmente risolvibile) è rappresentato dalla **generazione degli indirizzi multicast udp**, associati per ogni file. Per come è sviluppata la funzione di assegnamento degli indirizzi non vi è riutilizzo ne riassegnazione di indirizzi non utilizzati perciò vi è un massimo di file creabili che ammonta a 16.581.375. Inoltre vi è la forte assunzione che gli utenti girino su di una rete locale.

Altri problemi possono essere quelli di una mancata implementazione di una funzione di terminazione nel Server o quella del trovare un valore ottimale per il Thread Pool, ma questo esula dalle specifiche richieste dalla consegna del progetto.

Guida Utente.

Manuale Client.

Il client è di semplice utilizzo, si serve delle classi ***Client.java***, ***ChatThread.java*** e dell'interfaccia ***SubscribeInterface.java*** (per la registrazione via RMI).

Per una corretta esecuzione basterà reperire i file sopra citati per poi compilare da terminale la classe Client ed eseguirla:

1. `javac Client.java`
2. `java Client`

All'avvio il programma si presenterà come un qualsiasi programma dotato di CLI (es. Bash su Linux) e rimarrà in attesa di ricevere e processore tutti gli input utente, finché egli non digiterà il messaggio "exit", comando che appunto farà terminare il processo Client.

La lista dei comandi è consultabile digitando il comando turing –help che mostrerà la lista dei comandi possibile. L'utilizzo del programma resta comunque abbastanza intuitivo, infatti l'utente verrà guidato tramite stampe a schermo circa il comportamento di TURING, quindi verrà notificato con messaggi di risposta o errore in base all'input digitato.

Sintassi base dei comandi è: turing nomecomando <paramentri>. Viene di seguito riportata la lista dei comandi completa supportata da TURING:

```
commands:
      register <username> <password> | Registra l'user
      login <username> <password>    | Connette l'user
      logout                         | Disconnette l'user

      create <doc> <numsezioni>      | Crea un documento
      share <doc> <username>         | Condivide un documento
      show <doc> <sec>               | Mostra una sezione del documento
      show <doc>                     | Mostra l'intero documento
      list                           | Mostra la lista dei documenti
      edit <doc> <sec>               | Modifica una sezione del documento
      end-edit                       | Fine modifica della sezione del documento

      send <msg>                     | Invia un messaggio sulla chat
      receive                        | Visualizza i messaggi ricevuti sulla chat
```

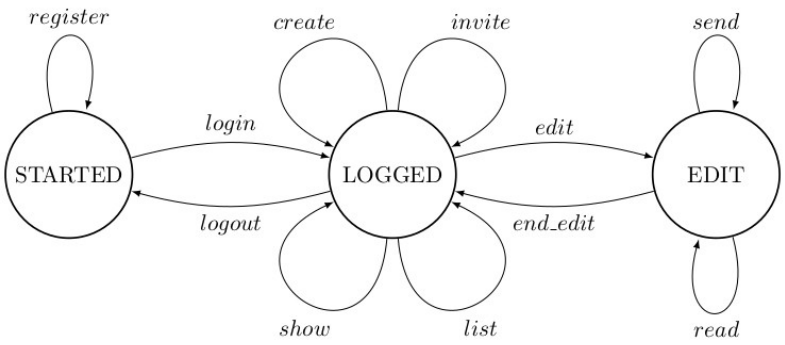


Figura 3: Automa a stati finiti TURING lato Client

È bene ricordare, che l'utente connesso a Turing tramite il Client fornito potrà spostarsi solo in uno dei 3 stati dell'automa mostrato in figura 3.

Quindi un utente non potrà inserire arbitrariamente i comandi, ma solo quelli concessi dall'automa, ovvero quelli che consentono una naturale esecuzione del programma. In ogni caso l'utente viene notificato se il comando inserito è lecito, oppure meno, invitandolo a scrivere uno dei comandi più opportuni.

Manuale Server.

Per far girare il Server sarà necessario possedere i seguenti file: ***ClientHandler.java***, ***FileData.java***, ***FilesDB.java***, ***Server.java***, ***SubscribeInterface.java***, ***UserData.java*** *e* ***UsersDB.java***, per poi andare a eseguire i soliti comandi di compilazione ed esecuzione per quanto riguarda il file Server.java:

1. `javac Server.java`
2. `java Server`

Non occorre inserire nessuno parametro in quanto è tutto codificato all'interno del file Server, basterà semplicemente non aver nessun servizio attivo alla porta **6666** per quello che riguarda il ServerSocketChannel, e alla porta **8556** per quanto riguarda il servizio di registrazione via RMI.