TURING:

disTribUted collaboRative edItiNG

Progetto Laboratorio di Reti

A.A. 2018/19

Michele Morisco N° Matricola: 505252 Corso A

INTRODUZIONE

Il progetto prevede l'implementazione di TURING (disTribUted collaboRative edItiNG), ovvero uno strumento per l'editing collaborativo di documenti.
Per realizzare questo strumento si è pensato inizialmente di creare una classe Client, che gestisce l'interfaccia utente (CLI) di TURING (*MainClassTuringClient.java*) e una classe Server che gestisce sia le richieste dei vari client che le operazioni sui servizi che offre TURING (*MainClassTuringServer.java*).
In principio, a lato server viene aperta una connessione tramite un thread che esegue tutte le operazioni necessarie per aprire una connessione e gestisce le richieste dei vari utenti.
Questo thread esegue la classe Runnable *ServerTuring.java*, ovvero il vero "server" dell'applicazione.
Il server si è deciso di progettarlo in **NIO** poiché esso è buffer-oriented e, soprattutto, non è una IO bloccante.
Dunque, verranno utilizzati i SocketChannel che aprono una connessione e attendono i messaggi di richiesta da parte dei client.
Si è poi utilizzato il **selettore** per gestire più connessioni di rete che verranno gestite da un unico thread; si è scelto questo approccio perché ci consentirà di ridurre il thread switching overhead.
Dopodiché il server delega la richiesta pendente come un task (*Task.java*) ad un pool di thread precedentemente dichiarato. Il pool di thread è stato dichiarato senza un vincolo sul numero di thread concorrenti.
Si è adottato l'utilizzo del **thread pooling** per ridurre ulteriormente il thread switching e di incrementare le prestazioni del programma.
I messaggi e le richieste che si inviano tra il server e i client sono oggetti di tipo **JSON**, essi contengono tutte le informazioni necessarie per eseguire un comando di TURING nel caso del server oppure di leggere il riscontro di tale comando nel caso del client.
La classe *Task.java* contiene tutte le funzionalità base di TURING (registrazione nuovo utente, creazione nuovo documento, modifica documento, ecc...).
Quando essa riceve l'oggetto JSON inviato dal client, esegue il **parsing** di tale oggetto, dopodiché leggerà nell'intestazione il tipo di comando che il client ha fatto richiesta, in questo modo deciderà quale funzionalità eseguire.
Dopo aver eseguito la funzione, il server prepara un **messaggio di risposta** per l'utente che ha inviato la richiesta con l'esito dell'operazione.
Invece, a lato client la classe *MainClassTuringClient.java* gestisce l'interfaccia utente in linea di comando sul terminale. Al suo interno si ha un ciclo infinito che chiede un input all'utente; dopodiché controlla tramite una serie di *if_then_else* la sua correttezza e quale operazione abbia scelto, altrimenti se errato, solleva un'eccezione di cui stampa l'errore a video.
I comandi disponibili di TURING sono elencati nell'ultima pagina.
La classe *ClientTuring.java* viene attivata come thread dalla *MainClassTuringClient.java* che permette di attendere una routine di messaggi di risposta da parte del server.
In questa classe inizialmente, si imposta un nuovo socket per la connessione con un numero di porta e un indirizzo IP speciale 127.0.0.1, tale indirizzo viene di norma usato dalle applicazioni per comunicare con lo stesso sistema su cui sono in esecuzione, ovvero in locale per fare testing.
La seguente classe attende una risposta dal server che arriva dopo aver mandato una richiesta; dopodiché, la classe legge le risposte che sono oggetti JSON. Dopo aver eseguito il **parsing**, controlla a quale tipo di operazione si è ricevuta la risposta. La maggior parte dei metodi in *ClientTuring.java* stampano a video l'esito del messaggio, solo alcuni (per esempio l'operazione edit o end-edit) eseguono altre operazioni che verranno discusse nel dettaglio in seguito.

STRUTTURA DATI

La struttura dati per la gestione dei documenti e degli utenti si è pensato di creare due **ConcurrentHashMap**: una per gli utenti e l'altra per i documenti. Si è scelto l'utilizzo di questo tipo di Map per garantire la concorrenza, ma questo aspetto verrà discusso in seguito.
Inoltre, si è realizzato due classi *User* e *Document*.
La prima si occupa delle informazioni di un utente come il nome, la password, il socket con cui si connette al server e lo stato della sua connessione (online o offline). Inoltre, al suo interno, si è definito alcuni metodi utili per la gestione degli utenti come cambiare lo stato della connessione, settare oppure chiudere il socket. Inoltre, per gestire le notifiche degli inviti di uno specifico utente si è dichiarata una lista di tipo String con i relativi metodi di gestione di una lista il quale verrà discussa in seguito durante la condivisione dei file.
Invece, l'oggetto *Document* prevede la gestione delle informazioni riguardo agli eventuali documenti creati. In questa classe si è creato metodi, per esempio come cambiare lo stato della fase di editing, interpretato da una lista di dimensione uguale al numero delle sezioni create per quel documento.
La lista di tipo String contiene gli username dei collaboratori (o del creatore) che stanno **editando una sezione**, infatti la posizione della lista corrisponde ad una sezione del documento.
Se appare un username in quella posizione della lista significa che tale utente sta editando quella specifica sezione e verrà concettualmente bloccata; altrimenti, se è una stringa vuota significa che nessuno la sta modificando.
Si è creato anche una lista per i collaboratori del documento per **memorizzare** tutti gli username degli utenti che **condividono lo stesso documento**.

GESTIONE THREAD

I thread principali attivati sono:

- il thread del server (*ServerTuring.java*)
- i thread degli *n* client online.

Nella classe *ServerTuring.java* si attivano al più *n* thread contemporaneamente dal **threadpool** (il numero degli utenti online che inviano contemporaneamente una richiesta al server) grazie al metodo *newCachedThreadPool()* fornito dalla classe *Executors*. Grazie a questo metodo si crea un thread pool che crea nuovi thread in base alle esigenze, ma che verranno riutilizzati se disponibili. Inoltre, i thread che **non vengono usati** per un certo periodo verranno **cancellati** e **rimossi dalla cache.** In questo thread pool, verranno eseguite le operazioni della classe *Task.java*.
Infine, nella classe *ClientTuring.java* quando l'utente sta modificando una sezione del documento viene attivato un thread per la ricezione dei messaggi della chat per poi rimuoverlo quando finisce la fase di editing.

GESTIONE CONCORRENZA

Per gestire la concorrenza nel progetto come è già stato anticipato prima, si è scelto di usare la **ConcurrentHashMap**, questa Map permette di gestire la concorrenza e rendere le **HashMap thread-safe**. L'approccio di questo tipo di Map è nato per gestire i casi di creazione di un nuovo utente o documento; infatti, se due o più utenti volessero registrarsi con lo stesso nickname allo stesso tempo (oppure due o più utenti volessero creare un documento con lo stesso nome) grazie al metodo fornito da questa Map: *putIfAbsent(K key, V value)*, è possibile inserire un nuovo elemento alla chiave K (se non esiste) in modo atomico, **evitando di creare** due utenti/documenti con lo stesso nome.
Invece, nelle classi oggetto *User* e *Document*, alcuni metodi sono *synchronized* dunque **quando un thread entrerà** in uno di questi metodi della stessa istanza dell'oggetto**, acquisirà la lock**, impedendo ad altri thread di eseguire il metodo stesso o altri appartenenti al solito oggetto finché tale thread non rilasci la lock quando esce dal metodo.
Essendo operazioni brevi e non molto costose questo approccio può funzionare senza bloccare un thread per troppo tempo.
Per gestire la **concorrenza** per il comando **Edit**, dato che può avvenire in caso in cui due o più utenti vogliano allo stesso tempo modificare una sezione uguale del proprio documento, si è utilizzato il metodo sincronizzato *changeEditState(String editor, int section)*. Questo metodo, quando un thread ci entra dentro, blocca l'intero metodo e controlla al suo interno se tale sezione non la sta modificando nessuno. In caso affermativo, viene **registrato** l'utente nella **lista sezioni** in fase di editing; quando il thread esce dal metodo rilascia la lock che verrà presa da un secondo utente che voleva modificare la stessa sezione. In questo modo, il secondo utente avrà una risposta negativa (poiché il primo utente è riuscito a "prenotarsi" prima) e verrà sollevata l'eccezione che la sezione è già in fase di editing.
Invece per gestire la **concorrenza** durante la **scrittura** o la **lettura di un documento** nel server si è pensato di creare una classe **ReadWriteServer.java**.
Quando si crea un'istanza di tale oggetto gli viene passato come argomento il percorso dove risiede il file; dopodiché per implementare la concorrenza si è pensato di usare la **ReentrantReadWriteLock** fornita da Java settando l'opzione **fair** che è utile nel nostro caso. Infatti, queste lock gestiscono la concorrenza tra i lettori e gli scrittori, ovvero ci possono essere più thread lettori che leggono lo stesso documento ma vengono bloccati se un thread scrittore sta scrivendo nel documento. Nella classe si è implementato i metodi *sendFile()*, che gestisce la lettura del file, e *writeFile(String msg, int size)* che gestisce la scrittura.
Per quanto riguarda, invece, la scrittura o la lettura di un documento nella cartella del client non è necessario gestire la concorrenza poiché, essendo solamente il client (e quindi un solo thread) a leggere e scrivere i documenti al suo interno, non c'è rischio che esistano uno scrittore e un lettore nel solito documento.

REGISTRAZIONE UTENTI

Lato server

La registrazione di un nuovo utente, visto che uno dei vincoli del progetto prevedeva di registrare l'utente tramite RMI, si è pensato di creare un'interfaccia *registerUserInterface.java* che contiene le dichiarazioni dei metodi per registrare l'utente tramite RMI che sono poi implementati nella classe *registerUser.java*.
Quando il server riceve la richiesta da parte del client, esegue il parsing dell'oggetto JSON ricavando l'username del nuovo utente e la password associata. Dopodiché, vengono passati tramite argomento al metodo *register (String username, String password)* dichiarato nella *registerUserInterface.java*; all'interno del metodo viene semplicemente creato un nuovo oggetto di tipo *User* e inserito nella Hashmap impostando come chiave l'username stesso dell'utente dato che il nickname deve essere univoco tra i vari partecipanti al servizio.

Lato client

Dopo aver concluso l'operazione, il server consegna al client che ha fatto richiesta, una risposta. Se il riscontro sarà positivo creerà una directory per l'utente dove risiederanno i documenti scaricati per visualizzarli o modificarli; altrimenti, stampa a video solamente il messaggio di errore per avvertire l'utente dell'esito negativo.

LOGIN/LOGOUT UTENTI

Lato server

Quando si riceve la richiesta di **login** da un client, il server controlla se l'utente è registrato al servizio. Dopodiché controlla se l'utente non sia già loggato, se questi controlli sono superati con successo, il server **assegna il socket** all'oggetto *User* corrispondente all'username che si è loggato. Questa azione permette al server di capire quale utente ha lo status "online" e, inoltre, dà un'identità al socket che invia richieste riconoscendolo con il relativo nome utente.
Inoltre, viene controllato se nell'oggetto *User* all'interno della lista *invites*, sono presenti notifiche di collaborazioni.
In questo modo, viene inviato la lista delle notifiche pendenti in **piggybacking** con la risposta (**ack**), il quale il client la gestirà singolarmente.
Per la richiesta di **logout**, al contrario, verrà assegnato all'oggetto *User* un elemento vuoto che determina lo status di offline di un utente. Infine, qualsiasi delle due operazioni il server invierà un ack con l'esito della richiesta al client.

Lato client

Dopo aver inviato la richiesta, client attende la risposta e dopo averla ricevuta, nel caso del **login**, controlla se ci sono notifiche pendenti ricevute in piggybacking con l'ack. Se fossero presenti allora le stamperebbe a video assieme al messaggio di esito positivo.
Invece, per il **logout** dopo aver ricevuto la risposta affermativa, essa viene stampata a video e dopodiché, viene **chiuso** il **programma** del client.

CREAZIONE DOCUMENTI

Lato server

Quando il server riceve la richiesta di creazione di un nuovo documento, esso crea inizialmente una cartella nella directory dove si immagazzinano tutti i documenti creati. La cartella avrà il nome del documento e ogni sua sezione corrisponderà ad un file di testo, come viene suggerito nelle specifiche del progetto.
Di conseguenza, viene creato un nuovo oggetto *Document* e inserito nella Hashmap corrispondente impostando come chiave il nome del documento dato che esso sicuramente sarà univoco. All'oggetto *Document* viene **assegnato** un **indirizzo statico Multicast** per la chat, ma questo verrà discusso in seguito.

Lato client

A livello client, viene soltanto riscontrato l'esito dell'operazione come già discusso nella registrazione dei nuovi utenti.

CONDIVISIONE DOCUMENTI

Lato server

Per questa operazione, il server inserisce l'utente con cui si vuole condividere il documento in una lista collaboratori presente all'interno dell'oggetto *Document*. Tale lista ovviamente raccoglie tutti i nickname degli utenti che possono visualizzare/modificare il documento creato da un altro utente. Dopo che l'operazione è andata a buon fine, il server prepara un altro oggetto JSON, oltre alla classica risposta di esito positivo. Il messaggio è di tipo **invite**, ovvero la notifica all'utente con il quale si è voluto condividere il documento, dato che l'implementazione di notifiche per gli inviti sono una specifica del progetto.
In questo caso, possono accadere due scenari: un caso in cui l'utente è online e dunque, il server prepara immediatamente la notifica tramite un oggetto JSON inviandolo al destinatario con cui si è voluto condividere il documento, utilizzando il metodo *sendInvite(JSONObject msg, Socket s)*.
Altrimenti, in cui l'utente è offline il server salva l'oggetto JSON all'interno di una lista inviti all'interno di *User*. Dopodiché, quando tale utente farà il login, il server invierà tutte le vecchie notifiche, se presenti.

Lato client

Il client che ha fatto richiesta riceve soltanto la risposta del server, positiva o meno. Nel caso fosse positiva, l'utente che ha ricevuto l'invito di condivisione riceverà una notifica immediatamente se online; altrimenti, verrà ricevuta successivamente al suo login.

VISUALIZZAZIONE DOCUMENTI

Lato server

Un altro vincolo del progetto è quello di usare **NIO** nella gestione dei files. Quindi, per visualizzare un documento o una singola sezione, si è pensato di leggere il file (se bisogna visualizzare una sezione del documento) oppure tutti i file nella cartella (se bisogna visualizzare l'intero documento) utilizzando il metodo *sendFile()* nella classe *ReadWriteServer.java*.
Esso utilizza il metodo fornito da Java **transferTo(long position, long count, WritableByteChannel target)** che permette di **trasferire** byte dal canale del file a un dato canale scrivibile. Si è pensato di utilizzare questo metodo perché è molto efficiente rispetto ad un semplice loop che legge dal canale sorgente e scrive nel canale destinazione. Inoltre, molti sistemi operativi possono trasferire direttamente i byte da una cache del filesystem al canale che ci interessa senza copiarli. Dopo aver inviato il file tramite questo metodo, viene inviato un ack con l'esito al client.

Lato client

Il client prima di inviare la richiesta creerà una cartella (se deve leggere tutto il documento) all'interno della directory dei documenti scaricati dopodiché gestirà l'ack ricevuto dal server che sarà solamente un messaggio di risposta per quanto riguarda l'esito dell'operazione richiesta.

EDITING DOCUMENTI

Lato server

Per l'editing si legge il contenuto della sezione che si vuole modificare con le stesse modalità descritte per la visualizzazione dei documenti. Poi si imposta, dopo tale operazione, che quella specifica sezione è in fase di editing inserendo il nome dell'utente che ha inviato la richiesta tramite il metodo *changeEditState(String editor, int section)*, esso inserisce l'username all'interno della lista delle sezioni della classe *Document* che si vuole modificare. In questo modo si **blocca** quella particolare **sezione** per quello **specifico utente** finché non verrà rilasciata dopo aver finito la fase di editing. Oltretutto, viene consegnato al client un **indirizzo Multicast** per permettere al client di utilizzare la chat tra i collaboratori che stanno modificando il documento e tale indirizzo verrà consegnato in **piggybacking** con l'ack del server. Questa funzionalità verrà spiegata nel dettaglio in seguito.
Quando riceve la richiesta di fine editing, il client passerà l'intero contenuto aggiornato sul file che salverà sovrascrivendo lo stesso file.

Lato client

Per quanto riguarda il client, invece, dopo aver ricevuto l'ack il file viene salvato nella **cartella Edit**, una cartella temporanea dove inserire i file da modificare. Dopo aver creato la sezione da editare, come spiegato nelle specifiche del progetto l'utente la può modificare tramite altri programmi esterni. Quando l'utente finisce l'editing, invia una richiesta end-edit, il quale il client eseguirà le stesse operazioni del server quando invia il contenuto del file.

CHAT

Una funzione di TURING è quella di offrire un piccolo servizio di chat durante l'editing di un documento e un vincolo del progetto nella realizzazione della suddetta chat era quello di implementarla tramite Multicast.

Lato server

Per realizzare ciò, si è pensato prima di dichiarare una variabile globale *AddressChat* in *ServerTuring.java*, essa rappresenta **l'indirizzo di tipo Multicast statico** che sarà assegnato ad ogni documento nell'atto della sua creazione. Si è scelto un approccio statico per semplificazione dato la natura del progetto, in altri casi si potrebbe assegnare l'indirizzo Multicast ogniqualvolta un primo utente vuole modificare una sezione del documento e toglierlo fin quando nessun utente lavora su quel documento rendendo nuovamente disponibile quell'indirizzo.

Malgrado questa semplice scelta progettuale, l'utente quando farà richiesta di modificare una sezione di un documento riceverà, oltre la sezione da modificare, l'indirizzo Multicast del relativo documento.

Lato client

Dopodiché il client attiva un thread tramite la classe **UDPChat.java** che inizializza il receiver dei messaggi che saranno ricevuti in Multicast dagli altri utenti in background. In questo modo potrà comunicare con gli altri utenti, se presenti, inviando messaggi tramite il metodo *send(String msg)* il quale al suo interno viene preparato il DatagramPacket con dentro il messaggio effettivo da inviare più l'orario di invio e aggiungendo nell'intestazione il nome del mittente.
Per ricevere i messaggi (che saranno ricevuti in modo asincrono, in questo caso), l'utente può utilizzare il metodo *receive()* che richiama un metodo della classe thread **UDPChat.java** che stampa a video tutti i messaggi inviati sulla chat, salvati in una lista. Inoltre, per evitare di ricevere i messaggi inviati a sé stesso si è utilizzato l'intestazione con il nome del mittente citato sopra. Questo permette di escludere nella lista dei messaggi da stampare, quelli inviati dall'utente che fa richiesta del metodo *receive()*.
Infine, quando l'utente finisce la fase di editing tramite il metodo *endEdit(String name, String section)* invoca il metodo della classe **UDPChat.java**, *chatStop()* che ferma il ciclo while e chiude il canale di chat per quell'utente. Con questa scelta progettuale, la chat di TURING si chiuderà quando tutti i partecipanti usciranno dalla modalità editing ma sarà un'azione eseguita dai vari client e non dal server stesso.

COMPILAZIONE ED ESECUZIONE

I file sono divisi in due cartelle, Server e Client.
Per poter compilare ed eseguire il progetto bisogna prima di tutto digitare nel terminale il seguente comando per entrambe le cartelle:

```
javac TuringException/*.java
```

Tale comando compila tutte le classi contenute nella cartella (package) TuringException, ovvero tutte le eccezioni personalizzate per questo progetto.
Dopodiché, si digita i seguenti comandi nell'ordine per compilare il client:

```
javac UDPChat.java
javac -cp ".;./json-simple-1.1.1.jar" MainClassTuringClient.java
```

Di conseguenza, si digita i seguenti comandi nell'ordine per compilare il server:

```
javac CommunicationInterface.java
javac -cp ".;./json-simple-1.1.1.jar" User.java
```

```
javac registerUserInterface.java
javac -cp ".;./json-simple-1.1.1.jar" registerUser.java
javac MainClassTuringServer.java
```
Successivamente, per eseguire il progetto si digita nel terminale i seguenti due comandi nell'ordine:
```
java -classpath ".;./json-simple-1.1.1.jar" MainClassTuringServer
java -classpath ".;./json-simple-1.1.1.jar" MainClassTuringClient
```
I comandi per eseguire le operazioni di TURING sono le stesse suggerite nelle specifiche del progetto:
```
$ turing --help
usage: turing COMMAND [ARGS ...]
commands:
register <username > <password > registra l'utente
login <username > <password > effettua il login
logout effettua il logout
create <doc > <numsezioni > crea un documento
share <doc > <username > condivide il 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 <doc > <sec > fine modifica della sezione del doc.
send <msg > invia un msg sulla chat dove il msg è tra "msg"
            (virgolette)
receive visualizza i msg ricevuti sulla chat
```
Dato che il progetto non adotta un sistema persistente, ogni chiusura del server comporterà la perdita di tutti i dati acquisiti durante la sua esecuzione.
Inoltre si consiglia di eliminare le cartelle DB_Document e ogni cartella creata per gli utenti registrati per una corretta esecuzione del programma.

Per questo progetto si è usato la libreria esterna JSON-Simple per implementare gli oggetti JSON.
Tale progetto è stato scritto con Eclipse e testato su un sistema operativo Windows.