Relazione per il progetto di Laboratorio di Reti

Francesco Bocchi 550145 Corso B

26 ottobre 2019

Indice

1 Introduzione

Il progetto richiede lo svilupppo di uno strumento per l'editing collaborativo di documenti, capace di offrire un insieme di servizi minimale per la loro modifica. L'impostazione segue il paradigma client/server e prevede allo scopo due classi, *TuringServer* e *TuringClient*, che rappresentano rispettivamente il lato server ed il lato client di TURING.
In particolare la struttura del client segue l'automa proposto nelle informazioni aggiuntive per il progetto, ovvero in un determinato istante il client potrà trovarsi in tre diversi stati, descritti nel seguito, in cui saranno possibili solo determinate operazioni.

2 Scelte progettuali

2.1 Strutturazione del codice e classi principali

Per la progettazione sono state realizzate diversi classi, cercando di rendere indipendente ogni modulo, seguendo quanto più possibile il principio di SoC, *separation of concern*.

L'unità più semplice modificabile è la sezione, rappresentata per l'appunto dalla classe **Section** che contiene il path della sezione e lo stato (un flag) che ne verifica la disponibiltà alla modifica.

Le sezioni sono contenute in documenti, gestiti mediante la classe **Document**. All'interno di questa troviamo un vettore immutabile che rappresenta le sezioni di cui è composto un documento, il creatore del documento, che insieme

al nome del documento, identifica univocamente questo nell'insieme di tutti i documenti gestiti dal server: a tale scopo supponiamo che l'utente *pluto* crei un documento di nome *mio*, allora nell'insieme dei documenti del server avremo un documento chiamato *mio_pluto*. In questo modo si permette la presenza di documenti con lo stesso nome, ma in realtà essendo il nome di un un utente univoco, la coppia nome documento e nome utente garantisce l'univocità del documento stesso.
La classe inoltre contiene il path del documento che ne identifica la posizione all'interno del server, un vettore che contiene i nomi degli eventuali utenti che collaborano alla gestione del documento ed infine ad ogni documento è associato un indirizzo di multicast, utilizzato per implementare il servizio di chat durante la modifica del documento.

Ogni utente viene gestito mediante la classe **UserData**. All'interno di questa troviamo un vettore di documenti di cui l'utente è creatore, uno di cui è collaboratore. Di particolare importanza sono il flag di notifica ed un SocketChannel utilizzato appositamente per le notifiche: quando un utente effettua il login al servizio, previa registrazione, viene attivato un thread Listener che sfrutta il SocketChannel per connettersi al server ed ascoltare gli inviti che vengono spediti a questo utente, in modo tale da notificare l'invito in tempo reale. Ad ogni utente quindi sono associate due socket: il server utilizza la prima per comunicare in maniera diretta con il client, mentre usa la seconda per notificare gli inviti al thread listener. In questo modo permettiamo la ricezione della notifica all'utente in maniera estemamente immediata. Si sarebbe potuto effettuare un controllo periodico da parte del client, ma oltre a non essere un meccanismo immediato (potrebbe passare del tempo dal momento in cui la notifica viene inviata a quando il client effettua il controllo), l'utente durante questo piccolo intervallo non potrebbe interagire con il servizio di turing.
Se l'utente è offline al momento della ricezione dell'invito, il server provvede ad impostare il bit di notifica in modo tale che al successivo login dell'utente, questo possa essere avvisato circa gli inviti ricevuti.

In riferimento agli stati in cui puà trovarsi il client, questo segurià il comportamento specificato in figura.

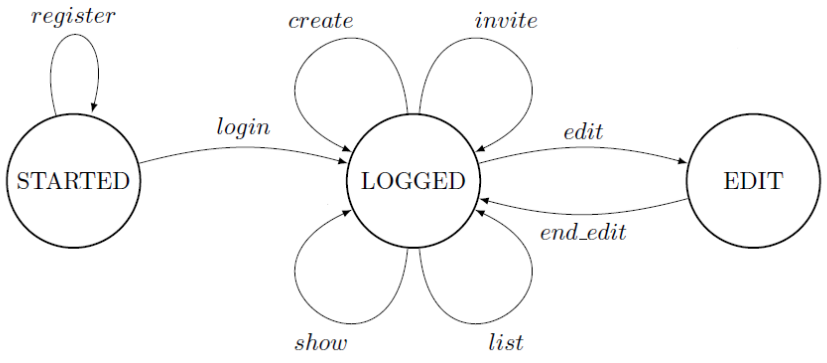

Figura 1: Descrizione grafica del comportamento del client Turing

Il client è definito nella classe **TuringClient**, ma in riferimento all'automa del comportamento atteso proposto, tale classe rappresenta lo stato iniziale del processo client, in cui è possibile eseguire solo le operazioni di registrazione e di login. Effettuato il login al servizio, in caso di successo, viene avviato un nuovo oggetto di tipo **ClientOperations**, che rappresenta lo stato logged, in cui sono possibili le principali operazioni offerte dal servizio turing, ma in cui non è possible eseguire altre registrazioni o altri login. Quando invece un utente richiede l'editing di un documento, passa in un nuovo stato in cui è possibile eseguire solo l'operazione di end-edit, che riporterà il client di nuovo nello stato logged. L'operazione di logout termina il servizio.
Scendendo nel dettaglio, nello stato started è possibile registrare gli utenti al servizio di turing attraverso il metodo RMI di registrazione offerto dal server oppure eseguire il login: è in questo momento che viene avviato il thread Listener descritto in precedenza che rimarrà in ascolto di nuovi inviti tramite una read sul socket secondario impostato in modalità bloccante.
Nello stato logged, identificato dalla classe ClientOperations, se si tratta della prima volta che il client effettua il login, vengono create tre cartelle che ospiteranno il lavoro effettuato dal client: la cartella dei documenti scaricati conterrà tutti i documenti che un client richiede con l'esecuzione del comando show, la cartella sezioni scaricate è analaoga alla precedende, ma contiene le singole sezioni ed infine è presente la cartella sezioni modificate che ospita le sezioni di cui il client ne richiede la modifica. Terminata la creazione delle cartelle il client inizia a gestire le richieste che l'utente immette tramite riga di comando.
Nello stato di edit il client può, tramite il suo editor personale, modificare la sezione scaricata, che sarà contenuta nella cartella *sezioni modificate*. In tale stato non è permesso eseguire altri comandi al di fuori di end-edit. Ogni altro tipo di immissione restituirà un avviso al client che lo informa circa questa impossibilità. Da notare che i parametri passati al comando di end-edit dovranno necessariamente corrispondere a quelli con i quali siamo entrati nello stato di edit. Non è permesso infatti terminare la modifca di una sezione diversa da quella che si sta modificando. Terminata la modifica, il client invia il testo modificato al server, il quale provvederà a sovrascrivere il vecchio contenuto con quello nuovo.

La classe **TuringServer** definisce il server. Per realizzare la comunicazione con gli altri client viene effettuato il multiplexing dei canali NIO mediante l'uso di un selettore, in modo da evitare overhead dovuti a thread switching. Una volta avviato il server , viene creata una cartella *Documenti* che conterrà i vari documenti creati dagli utenti. Inizializzato il servizio di registrazione remoto, il server entra nella fase di multiplexing di richieste provenienti dai client. Di seguito è descritto in maniera riepilogativa come il server gestisce le richieste che accetta:

- 1. Se la chiave fa riferimento ad un nuovo client che si connette per la prima volta, semplicemente si configura il socket relativo a questo client in modalità non bloccante (per non sospendere l'attività del server durante le operazioni di I/O), si associa un array che verrà utilizzato per gestire gli esiti delle varie operazioni richieste ed infine si registra questo nuovo client in modalità *OP_READ*.
- 2. Se la chiave è pronta per la lettura si effettua il parsing del comando ricevuto e si esegue l' operazione richiesta. In particolare dopo che il

server ha effettuato i controlli necessari per poter eseguire l'operazione, il client viene registrato in modalità *OP_WRITE*, e nell'array che avevamo associato nella fase di accept vengono inseriti gli esiti dell'operazione richiesta.

- 3. Se la chiave è pronta per la scrittura allora vengono inviati gli esiti delle operazioni richieste. A questo punto il client viene registrato nuovamente in modalità *OP_READ*, in attesa di una nuova richiesta.

Al fine di mantenere traccia degli utenti registrati e di quelli online il server viene supportato dalla classe **ServerStructure** che mantiene la tabella hash degli utenti online, il vettore degli utenti registrati: attraverso questa classe il server può gestire tutti i documenti associati agli utenti.

Il serivizio di chat, realizzato mediante interfaccia grafica, è gestito dalla classe **ChatInterface**, in cui viene "disegnata" l'interfaccia della chat. Ogni volta che viene inviato un messaggio dagli utenti che stanno modificando un documento questo viene ricevuto da tutti i partecipanti con l'informazione su chi ha inviato quel messaggio. Inoltre ogni volta che un utente termina la modifica, nella chat degli utenti restanti compare un avviso che li informa su chi ha lasciato la chat. Essendo richiesta l'implementazione del servizio basata su multicast UDP, il server a tale scopo gestisce un vettore di **MulticastManager**: ogni oggetto di questo tipo è composto da due stringhe che identificano rispettivamente l'indirizzo di multicast associato ad un documento ed il nome del documento. L'assegnamento degli indirizzi di multicast è dinamico: ogni volta che un client esegue la modifica di una sezione del documento, il server controlla se esiste un indirizzo di multicast nell'array che non sia associato ad alcun documento. In caso positivo restituisce l'indirizzo di multicast individuato e salva la corrispondenza <*indirizzo di multicast, nome del documento*> nel vettore. In caso negativo il server provvede a generare il successivo indirizzo, inserendolo nel vettore ed associandolo al documento. La procedura ha lo scopo di riutilizzare gli indirizzi di multicast.

2.2 Strutture dati

Per la gestione degli utenti registrati al servizio viene utilizzata una ConcurrentHashMap, dove ogni entry può essere vista come una coppia <*Nome dell'utente, Oggetto UserData*>.
L'utilizzo di una struttura concorrente risulta necessario in quanto più utenti possono, in uno stesso momento, richiedere la registrazione tramite il servizio RMI. Dato che l'invocazione di un metodo remoto sullo stesso oggetto potrebbe essere eseguita in maniera concorrente, l'implementazione dell' oggetto remoto necessita di essere thread-safe.

Inoltre la scelta di questa struttura dati è dovuta al fatto che se immaginiamo una mole di carico di utenti registrati molto alta, utilizzare ad esempio una lista avrebbe abbassato drasticamente l'efficienza della collezione, mentre attraverso la tabella hash, possiamo rapidamente accedere ad uno specifico campo, tramite la funzione di hashing.
Si è scelto l'utilizzo di una ConcurrentHashMap anche perchè permette l'overlapping delle operazioni sulla struttura, non bloccando tutta la tabella ad ogni

operazione. L'unica attenzione particolare richiesta dall'uso della struttura è la modifica della semantica di alcune operazioni, in particolare per la regsitrazione di un nuovo utente viene utilizzata la funzione *putIfAbsent* per realizzare in maniera atomica sia il controllo che verifica se l'utente è registrato o meno, sia l'effettiva registrazione.

Inoltre ogni utente che effettua il login al servizio Turing viene inserito in un Vector. Tale vettore rappresenta l'insieme degli utenti online in un dato istante.

2.3 Thread utilizzati e gestione della concorrenza

Come già descritto in precedenza, ad ogni client è associato un thread listener per la ricezione immediata degli inviti a nuove collaborazioni, mentre il server è supportato da un thread di hello che verifica periodicamente le connessioni dei client.
Non avendo realizzato un server multithreaded, l'unico problema di concorrenza avviene sulla socket secondaria associata a ciascun client.
Il server infatti, una volta pubblicato il servizio di registrazione, avvia un thread definito nella classe **HelloThread**, il cui compito è controllare ad intervalli di tempo regolari se un client è crashato durante l'utilizzo del servizio turing. Dato che vi sono situazioni particolari in cui un utente può subire un crash, ad esempio nella fase di editing, deve essere eseguito un codice di recovery, dunque per non appesantire l'operato del server riguardo il controllo di tali errori, viene utilizzato questo thread di Hello.
A tale scopo il thread tenta una scrittura su i socket secondari degli utenti online: se il client è online, tale messaggio arriverà al thread listener associato al client che semplicemente ignorerà il messaggio, ma se la scrittura fallisce allora il thread provvede a rimuovere dal vettore degli utenti online tale client. Sorge a questo punto un problema: cosa succede se il client crasha mentre stava editando una sezione? A tale scopo, dato che una sezione non può essere editata da nessun altro finchè rimane occupata, se non fosse stato gestito questo tipo di evento, avremmo avuto possibilmente sezioni non più modificabili. Per risolvere il problema nella classe UserData sono definite due variabili che rispettivamente riferiscono il documento e la sezione che sono in fase di modifica da parte di un determinato client. A questo punto possiamo avere due comportamenti:

- Se il client termina correttamente la modifica, il server provvede ad impostare tali variabili a null.
- Se il client crasha durante la modifica, allora il thread di hello quando sarà attivato proverà a scrivere sul socket ottenendo errore. Dunque provvederà a toglierlo dalla lista degli utenti online, ed inoltre attraverso queste due variabili potrà accedere al documento che il client stava modificando e renderlo nuovamente disponibile.

Risulta quindi necessaria una lock su tale socket perche può essere utilizzata sia dal thread di hello, ma anche dal server per inviare un eventuale notifica per un invito ad una collaborazione. È tuttavia necessaria anche una lock sulle variabili che permettono il recupero del documento che l'utente stava modificando, dato che sono modificate sia dal server quando l'utente inizia e termina in maniera normale la modfica di un documento, ma anche dal thread di hello per effettuare il recovery descritto sopra.

È da notare anche che se si verifica un crash in fase di editing, avendo realizzato l'assegnamento degli indirizzi di multicast in maniera dinamica, il thread di hello controllerà anche se dopo il crash è necessario eliminare l'assegnamento dell'indirizzo associato a tale documento o meno.

2.4 Classi secondarie

Oltre alle classi citate sopra, nel progetto è presenta anche un'interfaccia che estende Remote utilizzata per il servizo di registrazione remoto. L'implementazione definita in **RegistrationServiceImpl** permette di registrare un utente al servizio, controllando che il nome non sia già presente nella tabella.

3 Istruzioni per la compilazione ed esecuzione

Per la compilazione è necessario eseguire

$ javac TuringServer.java
$ javac TuringClient.java

Di seugito viene proposto un manuale per l'interazione con il servizio di turing. Per avviare il server ed il client eseguire

$ java TuringServer
$ java TuringClient

La sintassi di ogni comando è definita da **turing** *<comando> <parametri>*. Una volta avviato il client saranno possibili tre operazioni:

$ turing register <username> <password>, per richiedere la registrazione
$ turing login <username> <password>, per effettuare il login
$ turing exit, per uscire dal servizio

Effettuato l'accesso, saranno disponibili i seguenti comandi:

$ turing create <nome del documento> <numero di sezioni>, per la creazione del documento con le sezioni specificate
$ turing share <nome del documento> <nome utente>, per condividere il documento con l'utente specificato dal parametro nome utente

$ turing list, restituisce la lista dei documenti a cui si ha accesso, includendo i rispettivi creatori. L'informazione sui creatori è necessaria poichè per le operazioni successive è richiesto il nome del creatore di un documento. Attraverso questo comando si può quindi venire a conoscenza di questa informazione.

$ turing edit <nome del documento> <sezione> <creatore del documento>, per richiedere la modifica del documento

$ turing end-edit <nome del documento> <sezione> <creatore del documento>,

per terminare la modifica

$ turing show <nome del documento> <sezione> <creatore del documento>, per visualizzare una sezione

$ turing show <nome del documento> <creatore del documento>, per visualizzare un documento

$ turing logout, per uscire dal servizio

Per finire l'ultimo comando è quello di send, per inviare messaggi ai client che stanno modificando un documento, realizzato all'interno della chat tramite il pulsante di invio. Nell'interfaccia grafica della chat si ha un riquadro dove poter scrivere del testo ed inviarlo con il pulsante di invio, posto immediatamente sotto.

Nota: Il progetto è stato testato su Ubuntu 16.04 LTS 64-bit.