Progetto LPR: Turing

Lorenzo Beretta

Aprile 2019

1 Introduzione

Il progetto consiste nella realizzazione di *Turing*, un tool per l'editing di documenti da parte di un gruppo di utenti connessi a macchine diverse. All'interno di *Turing* si ha la possibilità di creare documenti, condividerne i privilegi di editing, scaricare un intero documento o solo alcune sue sezioni ed effettuare l'upload delle sezioni aggiornate. Il tutto avviene chiaramente con un sistema di "lock" che non permette a due utenti di editare una sezione simultaneamente, garantendo così la consistenza di ogni documento.

2 Architettura del Progetto

2.1 Struttura generale

Turing è costituito da un server centrale e più client che vi si connettono. Le connessioni client-server avvengono tramite protocollo TCP, fanno eccezione solo alcune notifiche (la cui mancata ricezione non compromette il servizio) che il server invia tramite UDP. I client si registrano inoltre al servizio utilizzando una RMI su una tabella presente nel server. Infine i client che lavorano sullo stesso documento possono comunicare tra loro attraverso una chat multicast i cui indirizzi sono gestiti in modo centralizzato dal server ma la cui implementazione è indipendente.

2.2 Server

Il server è composto da un thread listener ed un pool di thread worker. Il listener utilizza un selector sui cui sono registrati socket in modalità *non-*

blocking, di questi uno, il socket *welcome*, è registrato come "accept" mentre gli altri sono registrati "read". Dunque *welcome* serve per stabilire la connessione TCP mentre gli altri socket, una volta pronti per la lettura, verranno passati agli workers. Gli workers leggono dal SocketChannel la richiesta del client e la soddisfano, dopo di che se il client si è disconnesso (con una operazione logout oppure riattaccando) chiudono il socket ad esso associato, altrimenti lo aggiungono alla coda dei socket da riascoltare. All'inizio del ciclo di esecuzione del server tale coda viene svuotata e i socket in essa contenuti registrati nuovamente nel selector.

STRUTTURE DATI

- un **ThreadPoolExecutor** fixed pool adottato, in combinazione con il channel multiplexing, per cercare di ottimizzare la scalabilità.
- una coda **BlockingQueue** concorrente per gestire i socket.
- un **Selector**, un **DatagramChannel** e il **ServerSocketChannel** welcome costituiscono i canali con cui comunicare con i clients.
- le **ConcurrentHashMap** userMap e documentMap associano il nome di un utente (documento) al rispettivo oggetto di tipo User (Document).
- la **ConcurrentHashMap** socketMap associa ad un socket l'utente attualmente connesso attravero quel client.

Queste strutture dati sono accessibili anche dagli workers e userMap è condivisa anche con la **RemoteTableImplementation** che supporta la RMI. Come si può notare la concorrenza data dal thread pool è gestita utilizzando strutture dati condivise sincronizzate, e sfruttando il meccanismo della coda per riascoltare i socket che garantisce che al più un thread in ogni istante possa accedere ad un socket.

2.3 Client

Il client prende come argomento l'indirizzo IP del server se questo si trova su una macchina diversa, se invece non viene fornito alcun argomento il client contatterà il server sull'indirizzo di loopback. Il client implementa una CLI, effettua il parsing dei comandi, li comunica al server e legge le risposte. Per

fare ciò utilizza due Thread: il primo esegue il loop principale e gestisce le comunicazioni sincrone che utilizzano TCP, il secondo è in costante ascolto di notifiche UDP dal server. Vi è inoltre un terzo thread che rimane attivo fintanto che l'utente sta editando un documento e rimane in ascolto sulla chat associata a quel documento. La concorrenza è triviale in quanto ogni thread accede a dati diversi, eccezione fatta per l'unico metodo sincronizzato è "ChatListener.readMsgs()" che stampa i messaggi fino ad ora ricevuti. Nell'implementazione del client non vengono utilizzate strutture dati notevoli, tutto il lavoro di memorizzazione è scaricato sul server.

3 Protocolli di Comunicazione

Client e Server comunicano utilizzando TCP, UDP e RMI:

3.1 TCP

Per ottimizzare la scalabilità ho scelto di utilizzare **SocketChannle** *non-blocking* accoppiati con un **Selector** lato server. Al contrario il client deve gestire solo una connessione alla volta, ho quindi optato per la comunicazione sincrona utilizzando la modalità *blocking* in quanto scelta naturale per un protocollo domanda risposta. Il protocollo utilizzato per formattare i messaggi TCP in Turing prevede di inviare delle istanze di **Message**, ovvero messaggi costituiti da:

- **Header** = Tipo Operazione (4 bytes) + Lunghezza Payload (4 bytes)
- **Payload** = ByteBuffer (dato che si è usato NIO)

Il tipo di operazione è descritto dalla enum **Operation**.

3.2 UDP

La comunicazione UDP server-client si è introdotta in quanto non si potevano inserire le notifiche di invito nel normale scambio di messaggi TCP in quanto, restringendoci ad un solo client questo risulta sincrono. Dato che questo tipo di messaggi sono brevi, sporadici, ed il loro smarrimento non ha conseguenze si è optato per la soluzione più economica. Vi è dunque un thread dedicato nel client che sta in ascolto su un **DatagramChannel**.

3.3 RMI

Il server lega l'oggetto implementante la **RemoteTableInterface** al registro attraverso la key "REGISTER-TURING". Con ciò il client può utilizzare una normale procedura RMI per invocare il metodo "register(user, password)" e registrare i suoi dati nel server.

3.4 Multicast

Il server assegna ad ogni documento avente almeno una sezione occupata un indirizzo Multicast, ovvero nel range `[224.0.0.0, 239.255.255.255]`. Questo è l'unico compito del server in questa comunicazione in quando, una volta ottenuto l'indirizzo del gruppo, i client comunicano autonomamente utilizzando dei **MulticastSocket** in ascolto su un thread dedicato. Il TTL dei messaggi è impostato ad 1 in modo che questi non escano dalla rete locale.

4 Test

Durante la stesura del codice ho testato indipendentemente i vari moduli, una volta ottenuto un buon livello di funzionalità delle singole componenti ho effettuato i test lanciando server e client sulla stessa macchina ed utilizzando l'indirizzo di loopback. Successivamente ho reso il programma robusto rispetto alle invocazioni di comandi errati (rispetto all'automa di funzionamento fornito alla consegna) restituendo messaggi d'errore specifici all'utente. Infine ho testato con successo *Turing* su due macchine linux collegate tramite una LAN Ethernet.

5 Istruzioni

Compilare con il comando:

```
javac Server.java Client.java server/*.java client/*.java
                        common/*.java
```

Eseguire il server con:

```
java Server
```

Se "serverIP" è l'indirizzo IP del server in dotted quad eseguire il client con:

```
java Client serverIP
```

Se invece server e client sono sulla stessa macchina sarà sufficiente:

```
java Client
```