Turing

Massimo Puddu

Gennaio 2019

Contents

1 Introduzione

Il progetto si pone l'obbiettivo di sviluppare un servizio di file sharing.
Per lo sviluppo del progetto ho dovuto utilizzare il linguaggio di programmazione imposto dal corso di Reti e laboratiorio dell'università di Pisa, ossia java 8. Le specifiche del progetto prevedono di utilizzare Rmi per la registrazione, Tcp per la richiesta delle operazioni di un utente registrato e udp con indirizzi multicast per la gestione delle chat tra utenti che modificano nello stesso momento uno stesso file.

2 Scelte implementative

2.1 Server

Ho deciso che ogni documento nel sistema operativo viena rappresentato come una cartella e le sezioni sono file numerati da 1 a n sezioni, inseriti nella apposita cartella. Per fare questo faccio creare nel sistema operativo la cartella Documenti_Server (Attenzione faccio cancellare la certella Documenti_Server esistente precedentemente, e con essa tutti i file contenuti al suo interno), all'interno di questa cartella creo altre cartelle che rappresentano i documenti, e al loro interno inserisco le sezioni che possono essere editati da Turing. Per questa ragioni ogni documento ha un nome univoco e nessun utente può creare un documento con un nome già usato da un altro utente.
Per la gestione dei client ho deciso di adoperare il multiplexing perchè non volevo dedicare un thread a ogni utente che esegue il login, in modo da rendere il server molto più scalabile in base al numero di utenti. Il client e server interagiscono tra loro inviandosi messaggi che si suddividono nella seguente struttura:
1)Lunghezza del messaggio
2)Elemento dell'enumeratore
3)JsonObject
Il primo specifica il tipo di operazione da effettuare il secondo racchiude le informazioni per poter eseguire l'operazione. L'unica eccezzione a questa regola si ha quando il server invia il contenuto di una sezione al client. Il JsonObject viene implementato usando la libreria JsonSimple fornito tra i file consegnti.

2.2 Client

Ho deciso di implementare un'interfaccia grafica con javafx per il client. Essa si suddivide in fase di login, descritta nel file MainClient.java, e in fase di utilizzo, descritta in MainViewClient.java. In quella di login si può anche richiedere la registrazione, mentre nell'altra, l'interfaccia è fatta da una treelistview in cui si può selezionare un file e/o sezione e vederne le relative informazione, una volta selezionati. Se si clicca su edit sarà possibile solo vedere se si viene invitati per nuovi file e sarà visibile sulla destra la chat con cui sarà possibile interagire con le altre persone che stanno modificando altre sezioni dello stesso documento, e ovviemente sarà visibile anche una TextArea in cui si potrà modificare il file, da

questo momento sarà possibile solo cliccare su end edit visto che gli altri button saranno nascosti.

3 Strutture dati

3.1 Server

Per la memorizzazione degli utenti è stata utilizzata una concuncurrent hash map per andare incontro alle esigienze del rmi. La concurrent tree map prende come chiave una Stringa che contiene il nome dell'utente e come valore prende la classe Utente che contiene la password, i documenti per i quali è abilitato alla modifica e infine utilizzo una Selectiokey per indicare se un utente è online, perchè è comodo avere la sua SelectionKey a seguito della operazione di inviti nei gruppi.
Per una gestione veloce degli utenti online utilizzo una Tree Map. Questa treemap prende come chiave una String. Questa String sarà ricavata dalla SelectionKey attraverso il suo metodo toString e come valore la classe Online questa è una tripla che contiene il nome dell'utente, il documento e la sezione che sta modificando, se ne sta modificando una. Come ultima struttura il server usa un ArrayList per la ricerca e l'inserimento di documenti del server. A proposito dei documenti bisogna fare una distinzione. Il server e il client condividono la classe Documenti con il Client, questa classe contiene le informazioni base di un documento, come il numero di sezioni, il creatore e il nome del file. Il server utilizza la classe Documenti_server che estende la classe citata prima. Questa classe mi permette di astrarre le sezioni. Infatti Documenti_server incorpora la classe Sezione, insieme collaborano per la creazione del documento, La classe documento_server crea la cartella che rappresenta il documento e crea un oggetto sezione per ogni sezione. Quindi l'oggetto sezione si occupa di creare il file sezione nel sistema operativo e di gestirlo. Il server però interagisce con le sezioni attraverso l'oggetto contenuto nell'arraylist, per l'appunto il documento_server.
Per quanto riguarda le sezioni trovo utile tenere presente che esse contengono anche un informazione relativa al loro ultimo tempo di modifica per permettere a un client di non scaricare nuovamente un file nel caso esso abbia già una coppia aggiornata del file, ma di questo ne parlerò meglio a tempo debito.
C'è inoltre un ultima astrazione relativo agli Ip multicast questa mi permette di richiedere un ipmulticast in modo da poterlo utilizzare per le chat dei documenti.

3.2 Client

Il client ha due thread principali,escludendo quello che si occuppa della chat, il primo si occupata dell'interfaccia il secondo resta sempre in ascolto del server. Per farli comunicare utilizzo una SynchronousQueue<String>. Inoltre entrambi condividono un Treemap in cui si memorizza come key il nome del file che rappresenta la sezione e come value il tempo in cui è stato scaricato o modificato

il file.
Utilizzo infine una struttura dati di JavaFX che è un TreeItem nel quale posso memorizzare un documento e poterne ricavare i dati quando selezione il suo nome nell'interfaccia grafica.

4 Threads

Avendo utilizzato il multiplexing il Server non si trova mai a dover gestire una situazione di concorrenza al contrario del client, in quando si riceve un invito a un documento devo proteggere la treemap contenente i tempi di modifica dei documenti e la treetableview, in quanto potrebbero essere modificate da due thread diversi nello stesso momento, per farlo ho utilizzato una Reentrantrant-Lock come spiegato a lezione. Non c'è bisogno di utilizzare una lock quando si prende un elemento già selezionato dalla treetableview perchè questa aggiunge nuovi elementi sempre alla fine di essa, quindi non può creare problemi su un elemento già selezionato.

4.1 Server

Il server ha solo due thread uno che gestisce il multiplexing e l'altro è quello che si occupa di gestire una chat. Ovviamente di quest'ultimo possono esserci più istanze contemporaneamente. La chat viene avviata quando un utente prende una edit su una sezione del documento ed è l'unico utente ad averla richiesta, mentre si chiude quando l'ultimo utente che aveva la edit su una sezione del documento chiama la endedit chiudendo il multicastsocket dal thread che si occupa del multiplexing e quindi facendo partire un eccezzione nel thread che si occupa della chat.

4.2 Client

Il client ha in realtà tre thread, quello che si occupa dell'interfaccia, un thread-listener che rimane sempre attivo in attesa di messaggi da parte del server e il terzo si occupa delle receive dei pacchetti udp.
Quello che si occupa dell'interfaccia fa le richieste al server e poi attende la risposta mettendosi in attesa sulla SyncronisedQueue. Dopo che il client listener inserisce una stringa nella syncronized il thread si sblocca e aggiurna l'interfaccia grafica. Il thread receiver funziona nel seguente modo, resta su una read ad aspettare messaggi, quindo li riceve il messaggio può essere composto da un OP_Risposta o da un CLIENT_Operazione, nel primo caso questo thread può inoltrare direttamente il messaggio alla synchronized, altrimenti nel secondo caso questo thread deve prima fare altre operazioni come ad esempio scaricare un file, notificare dell'avvenuta ricezione di un invito, far partire il thread per i pacchetti udp etc...
Il terzo thread appunto resta in ascolto dei pacchetti udp e aggiorna la Textarea per poter visualizzare un messaggio quando arriva, come per il server si chiude

quando il threadlistener chiude il MulticastSocket. Questo Thread è presente nel file ClientListenerUdp.java.

5 Guida all'utilizzo e implementazione

In questa sezione vado ad affrontare come ho strutturato l'interfaccia, come utilizzarla e come ho implementato la varie operazioni, sia lato server che client, parlando prima della richiesta del client, per poi passare a come viene elaborata dal server e infine cosa fa il client accettata la risposta dal server. Per quanto riguarda la compilazione basta prima compilare Mainserver.java e successivamente MainClient.java includendo in entrambi il jsonsimple fornito nella consegna. Mentre per l'esecizione basta lanciarli senza aggiungere argomenti.

5.1 Apertura

All'apertura del client e server non passo da tastiera alcun input. Per prima cosa il client prova a connettersi al server, se qualche passaggio fallisce fa apparire un messaggio di errore all'utente che viene invitato a premere il tasto fine per provare nuovamente a far connettere il client.

5.2 Login

Questa schermata serve per accedere oppure registrarsi, un utente registrato inseririsce i proprio nome utente e password per poi premere il tasto Accedi, può anche premere invio quando si trovo la password field per far partire l'evento accedi, così facendo si invia al server una stringa composta da richiesta più un altra stringa in formato json che contiene nome utente e password. Il server quindi riceve questo messaggio e parsa la stringa json per poi ottenere appunto il nome e la password, a questo punto fa tutti i controlli del caso per poi rispondere al client con un OP_OK seguito da CLIENT_NOTIFICA se e solo se mentre l'utente era offline è stato invitato a editare qualche documento, mentre se l'utente ha sbagliato qualche dato riceve la risposta ERROR.

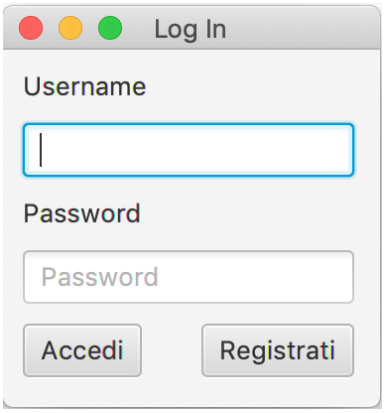

Figure 1: Schermata di login.

Si può anche premere il tasto registrati per far apparire la schermata successiva.

5.3 Registrazione

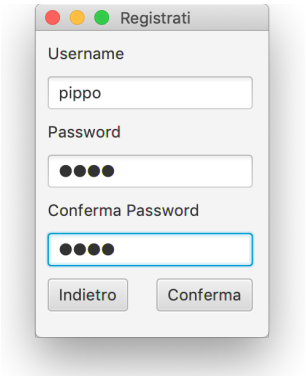

Figure 2: Registrazione

Ricordo che questa parte viene eseguita con rmi, quindi essendo già stata stabilita la connessione non mi resta che inviare la richiesta al registry per far eseguire l'operazione, come prima si può premere invio sull'ultima text field per far partire la registrazione. Quando Rmi esegui la registrazione aggiunge alla tabella hash l'utente e la password, è importante notare che non consensto di registrare nomi vuoti e più lunghi di 15 caratteri o password più corte di 4 lettere, questi controlli avvengono lato client.

5.4 Main View

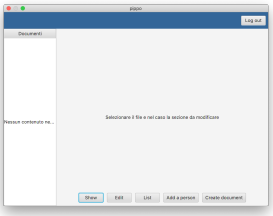

Figure 3: Schemata principale dell'applicazione

Schermata principale dell'applicazione, qui viene avviato un altro thread il ClientLiStener che si occupata di ricevere tutti i messaggi che vengono inviati dal server, in questo modo posso ricevere immediattamente gli inviti a nuovi documenti. In questo caso non ricevo una notifica vera e propria visto che la ho ritenuta fastidiosa, il nuovo documento viene quindi aggiunto nella treelistview presente alla sinistra dell'applicazione.

5.5 Create File

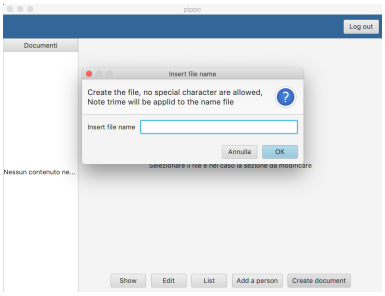

Figure 4: Crea documento

Quando si preme sull'apposito tasto apparirà una finistra di dialogo che chiederà il nome dal dare al file, non sono ammessi caratteri speciali nel file o nomi vuoti, dopo aver superato questo controllo apparirà un altra finestra di dialogo che chiederà il numero di sezioni, sono ammessi solo numeri da 1 a 99. Quindi si invia al server un messaggio contentente il tipo di operazione e il jsonobject che contiene le suddette informazioni. Il server quindi controlla che non ci siano già file con lo stesso nome e crea una cartella che rappresenta il nome del documento e dentro questa cartella inserisce i file che rappresentano le sezioni andando anche a creare all'interno dell'oggetto Documento_server gli oggetti sezioni che si occupano di una determinata sezione. Se tutto va a buon fine il Server manda un OP_Ok e ERROR o ERROR_ALREADY nel caso qualcosa sia andato storto. Quindi il client in caso positivo aggiunge il documento alla treeviewlist altrimenti ritorna un messaggio di errore apposito.

5.6 Share

Per poter inviare un invito bisogna prima selezionare il documento nella treeviewlist e in caso non sia selezionato apparirà un apposita AllertBox. Dopo che questo controllo se ne fa un altro in cui si guarda se il creatore del file corrisponde al nome della persona loggata e se entrambi vengono superati appare una textbox in cui inserire il nome della persona da invitare. Successivamente si invia al server il nome del documento che si vuole condividere e la persona che si vuole aggiungere il server controlla che la persona esiste andando a guardare nella tabella hash e prende la sua selectionKey da li, a questo punto controlla che l'invitato sia online e se lo viene aggiunto al suo attachment l'invito e il suo registro degli interessi viene cambiato in scrittura. In questo modo appena il server può invia l'invito all'invitato senza dover attendere che esso faccia una richiesta. Se l'invitato non era online viene invece settato un bit nella sua tabella hash per inviargli la notifica al prossimo login. L'utente che ha mandato l'invito riceve un OP_OK se tutto è andato a buon fine o oppositi messaggi di errore altimenti.

5.7 List

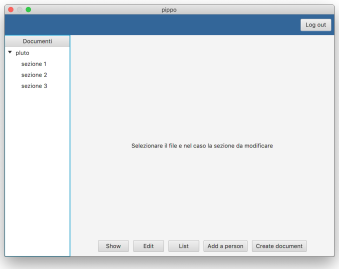

Figure 5: Lista

Il client invia una OP_LIST al server accompagnato da un JsonObject vuoto per comodità, Il server risponde mandando un JsonArray contenente tutti i documenti che l'utente è abilitato a modificare. Quando il client riceve il JsonArray controlla quali documenti ha già nella treelistview utilizzando la treeMap delle sezioni e inserisce appunto i documenti non presenti.

5.8 Show

Qua le cose si fanno più complicate, in quanto ho deciso di non far scaricare, ogni volta che il client lo richiede, il documento, in modo da evitare di dover mandare, nel caso peggiore giga e giga di file. Per prima cosa il client seleziona o la sezione o il documento che vuole visualizzare, nel primo caso invia al server un jsonobject sempre in formato di string che contiene il nome del documento, la sezione del documento e il tempo in formato di string che gli è stato fornito dal server l'ultima volta che ha richiesto un operazione di edit o show. Nel caso in cui si voglia visualizzare tutto il documento si clicca sul nome del documento e non su una delle sezioni e il server come prima riceve il nome del documento e nel campo sezione questa volta riceve -1 e gli vengono inviati tutti i tempi di tutte le sezioni, dove il tempo della sezione nel jsonobject ha come key proprio il numero della sezione. A questo punto il server ricevute queste informazione controlla inizitutto se i tempi delle sezioni coincidono con i propri e se lo sono non invia nuovamente la sezione, Sottolineo ora che il tempo è scandito da due variabili di tipo long, la variabile realtime viene incrementata ogni qualvolta qualcuno prende la edit o esegue una endedit che va a buon fine, mentre viene decrementata se la endedit fallisce o l'utente si disconnette mentre ha la edit, in quanto la modifica non è andata a buon fine e quindi il file rimane inalterato. L'altra variabile del tempo invece non tiene conto se un utente sta modificando e viene aggiornata solo quando un endedit va a buon fine. Quindi si può notare che quando qualcuno ha la edit la variabile realtime è dispari. Il server risponde al client mandando un CLIENT_SHOW seguito da un jsonobject che contiene tutti i tempi delle sezioni del server. Se uno o più dei tempi è dispari il client fa apparire un allertbox all'utente per fargli sapere quali sezioni sono in fase di modifica e sia in questo caso che se i tempi coincidono con quelli presenti nel server il client non cerca di scaricare di nuovo il documento, mentre se non coincidono e non è in fase di edit il client scarica la sezione e aggiorna i tempi di

questa. Il ClientListener inserisce i file scaricato nel file con il nome composto dal nome del documento più il numero della sezione. Dopo aver superato questa fase di download del file il client carica dal file alla textArea dove ogni sezione è separata da una newline.

5.9 Edit

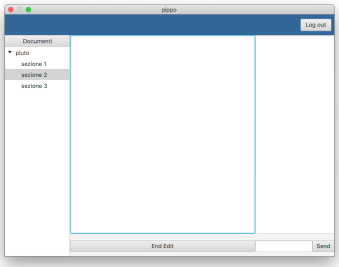

Figure 6: Si sta editando una sezione vuota, alla destra si può vedere la chat

Come per la Show il client e server si inviano i tempi ma questa volta il server manda al client un errore se un utente sta già editando la sezione, quindi il server invia solo la variabile del tempo e non il realtime, anche se è indifferente in questo caso. Fatti i controlli sul tempo il comportamento è praticamente analogo a quello procedente solo che il server invia l'operazione al client CLIENT_OPENUDP dove specifica ip multicast in cui si deve ascoltare o inviare messaggi alla chat, si apre quindi il terzo thread che si occupa di ascoltare questi messaggi. Arrivati a questo punto il client ha già scarito il file, se non aveva già una versione aggiornata e quindi carica il MainviewClient carica il file nella TextArea, ma questa volta la TextArea è modificabile e consente di editare il documento attraverso il client.

5.10 End Edit

Premuto questo tasto trasferisco il programma trasferisce il testo dalla TextArea al file. A questo punto il client invia un OP_ENDEDIT e poi invia come messaggio separato il file. Il server lo riceve e lo inserisce inizialmente su un file temporaneo in modo che se il client perde la connessione non ci si ritrovi con una sezione incosistente. Se tutto va a buon fine, chiude i FileChannel e rinomina i file in modo che il file temporaneo diventi il nuovo file della sezione di riferimento e il vecchio venga eliminato, fatto questo incrementa appunto la variabile del tempo e invia al client l'operazione CLIENT_CLOSEUDP o un ERROR_TOEDIT e il ClientListener a questo punto va a chiudere il thread che ascoltava i pacchetti udp.

5.11 Chat

Il server apre una nuova chat ogni volta che un utente tenta di editare un documento ed è l'unico a volerlo fare in quel momento. Viene quindi fatto, in questo caso, una richiesta a un oggetto che genera gli ip multicast restituendo l'indirizzo che verrà utilizzato per gestire una particolare chat. Il server quindi controlla l'indirizzo in particolar modo controlla che il multicast socket quando

si va a chiamare il metodo joingroup non lancia IOException, se invece accade si prova con un altro ip. Fatto questo si apre un nuovo thread che gestisce la chat per gli utenti che editano quella sezione. La chat verrà chiusa quando nessun utente sta più editando il documento. Il generatore degli ip usa gli ip sfruttati precedentemente.
Per quanto concerne il client apre un thread che ascolta i pacchetti udp in base all'ip che gli è stato inviato dal server quando ha richiesto la edit. Si possono nuovi messaggi premendo sul tasto di send o premendo invio sulla textbox. I messaggi più lunghi di 128 byte vengono suddivisi in più messaggi e viene specificato sempre chi li sta mandando.

5.12 Logout

Può essere eseguita in qualsiasi momento e può avvenire sia premendo sul apposito bottone o premendo sulla x. In quest'ultimo caso il server non manda però alcun messaggio di risposta e il client non richiede direttamente l'operazione. Nel primo caso il client manda un OP_LOGOUT e il client provvederà a rispondere. Una volta accettata la richiesta il cliente si ritroverà nella schermata di login per essere pronto a fare nuove richieste.

6 Conclusione

A causa dell'interfaccia grafica, ho deciso di non passare alcun argomento da terminale. Quindi per avviare al problema il server viene avviato sulla porta 5000, il registry del rmi viene creato e utilizzato sulla porta 9999, quindi il client deve connettersi a queste porte prima di poter fare qualsiasi richiesta. Per quanto riguarda i pacchetti udp vengono spediti dal client al server sulla porta 2000 e il client ascolta sulla porta 3000.
Per concludere ho ritenuto il corso di labaratorio molto utile, gli esercizi sono sempre stati molto utili e pertinenti a quanto spiegato a lezione rendendo lo sviluppo del progetto molto più immediato e semplice da capire, visto che avevano già risolto tutte le problematiche che avrei altrimenti incontrato sprovvedutamente al progetto.