Relazione TURING

Progetto del modulo di laboratorio di Reti di Calcolatori

Giulio Paparelli

546851 - corso B

a.a 2018/19

Indice

1 Architettura complessiva

1.1 Divisione in package

Per garanitire una modularita' alla applicazione si e' eseguita la seguente divisione in package

- server: classi relative al funzionamento del server
- client: classi relative al funzionamento del client
- sharedClasses: classi di utilizzo comune

1.2 Client

Il client di TURING si basa sulle seguenti tre classi

- Client.java
  - comunicazione con il server ed interazione con l'utente
- ChatHandler.java
  - gestione della chat multicast per i collaboratori di un documento
- TuringInviteHandler.java
  - gestione degli inviti a collaborare su un documento ricevuti da altri utenti

1.2.1 Thread usati

La parte client di TURING e' composta da 3 thread

1. Main Thread: quando si avvia il client il main thread entra in un loop, in cui
    a. attende l'input da parte dell'utente, controllando che
        i. l'input non sia vuoto
        ii. gli username, le password e i nomi dei documenti siano almeno lunghi una determinata quantita' (6 caratteri)
        iii. gli input numerici, come il numero di sezioni di cui e' composto il documento che si vuole creare, siano effettivamente degli interi
    b. chiama la funzione che gestisce il comando immesso

2. Thread di gestione degli inviti
    a. questo thread viene avviato non appena il client esegue con successo il login, mettendosi in costante ascolto di inviti. Quando viene ricevuto un invito a collaborare questo viene stampato. Il thread termina quando il client esegue il logout

3. Thread di gestione della chat
    a. il thread si avvia quando il client entra in fase di editing, cosa che comprta l'unione al gruppo multicast relativo al documento.
    b. Il suo scopo e' di rimanere in attesa di messaggi da parte degli altri collaboratori per poi salvarli in un `ArrayList`. Quando l'utente vorra' visualizzare la chat inserira' il comando `turing receive` che mostrera' a schermo tutta la cronologia dei messaggi.
    Inoltre il thread fornisce la funzione per l'invio dei messaggi.

1.3 Server

Il server e' implementato con il channel multiplexing, cosi' da monitorare se i vari channel relativi agli utenti sono pronti per una operazione I/O.

Quando si rileva la richiesta di connessione di un client il metodo accept() sulla socket del server crea una socket di comunicazione che viene quindi registrata nel selettore in modalita' lettura (OP_READ)

Quando si rileva un canale pronto per la read si procede con la lettura del messaggio inviato dal client e si gestisce la richiesta, modificando l'interestOps() della SelectionKey in OP_WRITE (il client dopo aver inviato una richiesta si mette in attesa della risposta)

- il protocollo di comunicazione viene spiegato meglio nel seguito

Quando viene rilevato un canale pronto per la scrittura si procede al recupero della risposta dall'attachment della selection key (risposta elaborata quando e' stata gestita la read da parte di quel client) e viene quindi spedito il messaggio.

1.4 Classi comuni al server e al client

Nel package sharedClasses sono presenti le classi che vengono usate sia dal modulo client che da quello server dell'applicazione.
In particolare abbiamo le seguenti classi

- ChatMessage.java
    - rappresenta il messaggio inviato nella chat multicast relativa ai collaboratori di uno stesso documento e fornisce i metodi di interazione con la chat (ricezione ed invio)

- Connection.java
    - contiene metodi statici che si occupano dell'invio e della ricezione dei messaggi che si scambiano client e server. I metodi sono appartenenti a due categorie
        - metodi per la ricezione di un messaggio
            - leggo un intero che rappresenta la lunghezza in byte del messaggio che sto per ricevere
            - leggo il messaggio
        - metodi per l'invio di un messaggio
            - invio un intero che rappresenta la lunghezza in byte del messaggio che sto per inviare
            - invio il messaggio

- Document.java
    - e' in pratica un descrittore del documento nel file system
    - contiene quindi il riferimento alle varie informazioni di interesse, quali
        - creatore
        - utenti autorizzati all'editing
        - numero sezioni
        - ...
        - porta su cui verra' aperta la chat multicast
    - in particolare fornisce un metodo statico per la lettura di un file, passando come parametro un oggetto di tipo Path

- DocumentSection,java

    - rappresenta le sezioni di cui e' composto un documento e fra le varie informazioni di interesse contiene il riferimento all'utente che attualmente sta editando quella sezione

- Message.java
    - rappresenta il tipo di messaggio che viene scambiato fra client e server, contiene tutte le informazioni che possono essere necessarie ad una determinata operazione.
    - una volta che sono stati popolati i campi necessari viene trasformato in JSON con la libreria GSON ed inviato al client o al server

- TuringValues.java
    - enumerazione di codici che rappresentano le varie operazioni e gli esiti associati

- User.java
    - rappresenta l'utente dell'applicazione e contiene le informazioni necessarie, fra cui
        - username
        - password
        - lista degli inviti a collaborare non ancora visti (inviti ricevuti quando si era offline)
        - stato attuale
        - lista dei documenti di cui l'utente e' il creatore
        - lista dei documenti di cui l'utente e' un editor autorizzato
        - socket per la ricezione degli inviti

2 Scelte progettuali

2.1 Assunzioni

Dato che un'applicazione di questo genere ha un numero elevatissimo di casi d'uso e di controlli necessari sono state fatte diverse assunzioni, cosi' da mantenere la complessia' e la leggibilita' del codice ad un livello ragionevole.

1. dato che il server non ha modo di mantenere i dati fra una sua esecuzione e l'altra si e' assunto che il server debba rimanere attivo, quindi il server non gestisce la sua terminazione, il che ha diverse implicazioni
    a. il server va terminato manualmente (ad esempio con ctrl^c, SIGINT)
    b. non viene gestita la cancellazione dei documenti risalenti ad esecuzioni precedenti, la cartella all_documents che contiene tutti i file del progetto va quindi eliminata manualmente ad ogni esecuzione del server
        i. questo e' necessario in quanto, usando sempre lo stesso percorso, se un utente con un username usato precedentemente creasse un documento con un

nome risalente alle esecuzioni passate verrebbe sollevato un errore (documento gia' esistente)

2. l'invito alla modifica di un documento con nome uguale ad un altro documento posseduto dall'utente invitato e' un caso non gestito
    a. l'utente invitato facendo l'editing prenderebbe sempre il suo documento

3. La scelta di creare un tipo Message unico per tutte le operazioni e' stata possibile grazie all'utilizzo di GSON e al JSON. Ogni operazione utilizza solo un certo numero di campi di un oggetto di tipo Message, lasciando nulli gli altri. Questo pero' non e' un problema in quanto l'oggetto di tipo Message viene trasformato in una stringa in formato JSON, nella quale i campi nulli semplicemente non vengono inseriti. Inviando la stringa JSON al server non si ha quindi uno spreco di risorse (banda) nel mandare campi non valorizzati. Una volta che il server riceve la stringa in JSON ricostruisce l'oggetto tramite GSON

4. la comunicazione degli inviti non viene gestita dal selettore (non viene testato isWritable() sulla socket in attesa sul client) in quanto lo specifico thread del client rimane in costante attesa di inviti ed e' praticamente certo che sia possibile scrivere senza problemi

5. il numero di sezioni di cui un documento e' composto parte da 0, quindi creare un documento con 4 sezioni produrra' i file
    a. 0.txt
    b. 1.txt
    c. ...
    d. 3.txt

2.2 Multiplexing

Per la gestione di molti utenti si e' optato per una soluzione con Multiplexing, al posto di una architettura multithreaded.
In questo modo, tramite un selettore, e' possibile gestire molte richieste senza appesantire il progetto con strutture dedicate alla concorrenza.
Come visto sopra, quando il selettore rileva una SelectionKey associata ad una socket pronta per un'operazione di I/O si controlla che questa sia:

- accettabile: actualKey.isAcceptable()
    - richiesta di connessione da parte di un client: viene creata una nuova socket di comunicazione ed impostata in modalita' non blocking ed associata ad una nuova SelectionKey in lettura, quindi la key viene registrata nel selettore

- leggibile: actualKey.isReadable()

    - richiesta di invio di un messaggio da parte di un client: si esegue la lettura per ottenere l'oggetto di tipo Message inviato dal client, acutalKey viene poi messa in scrittura (OP_WRITE) e si gestisce l'operazione richiesta
- scrivibile: actualKey.isWritable()
    - e' stata gestita una richiesta, si recupera la risposta prodotta dal server dall'attachment della chiave, quindi si risponde al client e si ritorna in modalita' lettura (OP_READ)
        - si suppone che un client eseguira', prima o poi, un'altra operazione

2.3 Protocollo di comunicazione

Come detto client e server comunicano con dei messaggi di tipo Message. Ad ogni operazione gestita da TURING sono associati due messaggi

- richiesta da parte del client
- risposta da parte del server

Per quanto riguarda il client notiamo che

- per motivi di semplicita' i controlli di sicurezza e di correttezza che sono effettuabili sul client sono effettuati sul client
    - username, password ed altri input sono stringhe valide o interi
    - stato del client
        - una volta che il server ha confermato il login viene valorizzata a true la variabile clientIsOnline
        - nelle successive operazioni verra' testata la variabile del client clientIsOnline e non verra' eseguito il controllo nel server
        - questo stesso metodo viene applicato diverse volte (clientIsEditing, ...)

- il client in fase di login invia un secondo messaggio con codice "Hello Invite" al server, cosi' da risvegliare la relativa SelectionKey e permettere al server di salvare nell'oggetto User la socket per gli inviti. Si assume che questa socket e' sempre pronta per la ricezione e quindi non viene aggiornato l'interestOps e registrata nuovamente nel selettore.

Il protocollo di comunicazione segue questo schema

1. Il client si connette
    a. viene creata la socket di comunicazione
2. il client calcola la lunghezza in byte del messaggio e la invia
    a. cosa possibile in quanto e' nota a priori la quantita' di byte da inviare: un intero in Java occupa 4 byte
3. il selettore nel server rileva un canale pronto alla lettura
4. il server legge la lunghezza in byte del messaggio che deve ricevere
5. il server legge il messaggio

    a. si legge esattamente il numero di byte comunicato precedentemente
6. il server registra la selection key in scrittura, genera il messaggio di risposta per il client e lo inserisce nell'attachment della selection key
7. il selettore nel server rileva un canale pronto alla scrittura
8. il server recupera l'attachment dalla selection key, quindi invia la lunghezza in byte della risposta al client, infine invia il messaggio di risposta

2.4 Flusso di esecuzione

1. si avvia il server
2. si lanciano i client
3. i client si registrano
4. i client eseguono il login
    a. il client avvia un altro thread in attesa di inviti ad editare documenti
5. i client eseguono operazioni varie
6. i client chiedono di editare un documento
    a. viene avviato un thread che gestisce la chat multicast fra gli utenti che editano in quel momento lo stesso documento
    b. il thread viene terminato quando il client termina l'editing
7. ...
8. i client eseguono il logout
    a. causa la terminazione del processo client

3 Esecuzione del progetto

Al lancio dell'applicazione viene creata la cartella che TURING utilizza per memorizzare i file e le sezioni.
Questa si posiziona allo stesso livello delle cartelle relative ai codici ed e' strutturata come segue

- all_documents
    - client: cartella dei documenti relativa al client
        - $utente_1$: cartella relativa ai documenti di un utente
            - $documento_1$: cartella delle sezioni di un documento
                - sezione in editing al momento
            - $documento_2$
            - ...
        - utente2
    - server: cartella dei documenti relativi al server
        - $utente_1$
            - $documento_1$
                - tutte le sezioni
            - ...

    - utente$_2$
    - ...

La compilazione avviene avviando lo script `compile.sh` (in ambiente unix)

Per avviare il progetto si eseguono i seguenti comandi posizionandosi nella cartella `turing`

- `java -cp ".:./gson-2.8.5.jar" server/Server`
- Si avvia una nuova finestra del terminale e sempre dalla cartella `turing` e da terminale
    - `java -cp ".:./gson-2.8.5.jar" client/Client`
- stesso procedimento per altri eventuali altri client
- si ricorda che ad ogni nuova esecuzione del server va eliminata la cartella `all_documents`

Le istruzioni di utilizzo di Turing sono visibili avviando il client, dando il comando `turing --help`