RELAZIONE PROGETTO RETI DEGLI ELABORATORI 2018/2019: TURING

A cura di

GRAMAGLIA GERLANDO 530269

CORSO B

INDICE

1.INTRODUZIONE

Il progetto comprende l'implementazione sia del Server che del Client. Il server opera effettuando il multiplexing dei canali tramite NIO, il suo "main" è contenuto nel "MainClassTuringServer" mentre "ServerMethods" contiene un insieme di metodi statici su cui il server si appoggia, principalmente, per inviare dati agli utenti. Il "main" del Client si trova, invece, in "MainClassTuringClient" e come per il server utilizza un insieme di metodi statici contenuti in "ClientMethodsTuring" per inviare e ricevere dati al/dal server. Il Client, inoltre, può avviare due Thread istanziando "Notify" e "ThreadChat" sotto determinate condizioni. La fase RMI, prevista per la fase di registrazione di un utente, si basa sull'utilizzo di "Registrazione e RegistrazioneImpl" e "UserInt e User". Le classi "Documento" e "UtenteOnline" vengono istanziate soltanto dal Server, queste, contengono e gestiscono le informazioni e le operazioni che riguardano i documenti creati dagli utenti e gli utenti stessi quando sono online.

2.CLIENT

a. MainClassTuringClient

Il programma inizia analizzando i parametri di args[], è possibile passare fino a 3 argomenti nel seguente ordine: numero di porta del registro, numero di porta con cui aprire il socket verso il server e il nome dell'editor che si vuole utilizzare; per default, queste variabili, assumo i valori di "9999", "5000" e "Notepad.exe" ed è, quindi, opzionale effettuare il passaggio dei parametri. Tra le altre variabili usate dal client troviamo

```
ByteBuffer buffer = ByteBuffer.allocate(1024); (riga 42)
```

Utilizzato per tutte le operazioni di scrittura e lettura che vengono effettuate nella comunicazione TCP con il server; mentre per leggere l'input dell'utente viene utilizzato

```
Scanner input = new Scanner(System.in); (riga 44)
```

Dopo questa fase preliminare si entra in un ciclo while dal quale si può uscire solo quando cade la connessione verso il server oppure quando l'input contiene il comando **`termina`**, in entrambi i casi viene settato il flag **`endclient=true.`** All'inizio di ogni ciclo si utilizza lo scanner sopra esposto per poter leggere l'operazione richiesta dall'utente, tale operazione deve essere scritta in un'unica riga, su di essa si effettua un'operazione di split che divide la riga in un array di stringhe. Una richiesta tipica si presenta nella forma:

<Tipo richiesta> <param1> <param2>

Dove **<parma1>** e **<param2>** devono comparire solo con determinati tipi di richiesta.

Il passo successivo è lo switch del primo elemento dell'array (nella posizione 0 troviamo sempre il <**tipo di richiesta**> dell'utente), i casi possibili sono:

- `case "register":` Si vuole effettuare la registrazione di un utente, per farlo è necessario che **<param1>** e **<param2>** siano diversi da `null` e che rappresentino rispettivamente Nome e Password. Se tale condizione viene soddisfatta andiamo a cercare il registro corrispondente attraverso il numero di porta del registro definito all'inizio del nostro programma ed invochiamo il metodo necessario alla registrazione, questo ci restituirà un valore booleano che indica l'esito dell'operazione.

- `case "termina":` Si pone `endclient=true` e il client termina.

- `default:` Si stampa un messaggio speciale.

- `case "login":` Si stabilisce una connessione TCP con il server e si invia la richiesta di login usufruendo del metodo statico

  `ClientMethodsTuring.sendOp(…) (riga 104)`

  Questo restituirà un valore che indica l'esito dell'operazione, se questa è andata a buon fine allora avvio un Thread

  `Thread threadnotify = new Thread(notifiche); (riga 114)`

  Dove `notifiche` è un'istanza di **Notify** (se ne parlerà in seguito). Adesso si entra in un while annidato al precedente dal quale è possibile uscire effettuando l'operazione di logout o se cade la connessione verso il server, all'inizio di ogni ciclo vengono effettuate le istruzioni descritte nel ciclo più esterno, quindi attraverso uno switch si va ad esaminare la richiesta dell'utente:

  - `case "logout":` per effettuare il logout

  - `case"create":` per creare un documento, si effettua dapprima un controllo del numero e del formato dei parametri inseriti se questi vengono soddisfatti allora si prosegue con l'operazione altrimenti si stampa un messaggio speciale a video e si ricomincia il ciclo in cui il client si mette in attesa dell'input. Nel caso i parametri siano corretti si invia la richiesta al server invocando il metodo statico

    `ClientMethodsTuring.sendOp(…) (riga 143)`

  - `case "list" :`viene richiesta la lista dei documenti invocando il metodo statico, che ritorna l'esito dell'operazione

    `ClientMethodsTuring.getlist(..)`

  - `case "share":`Si controlla il numero di parametri inseriti nell'input, se

viene soddisfatto allora si può procedere alla richiesta di share avvalendosi sempre del metodo `ClientMethodsTuring.sendOp(..)`

- `case "show":` si controlla il numero di parametri, in questo caso abbiamo due casi: l'input contiene solo il nome del documento (quindi vengono chieste tutte le sezioni) oppure può contenere anche il numero di una sezione specifica. Da notare che viene usato lo stesso metodo del caso list, questo perché in entrambi i casi stampiamo a video i dati ricevuti dal server.

- `case "edit":` Dopo i soliti controlli sull'input si effettua la richiesta di editing utilizzando il metodo

  `ClientMethodsTuring.requestedit(…)`

  In base al valore restituito individuiamo l'esito dell'operazione e il conseguente comportamento che il programma deve assumere. In caso di successo si aprirà il nostro editor con il contenuto della sezione richiesta e il client continuerà richiedendo di essere aggiungo alla chat mediante il metodo

  `ClientMethodsTuring.getIPChat(…)`

  Viene avviato un thread che si occupa di ricevere i messaggi che arrivano nella chat e di inserirli nella struttura dati `ArrayList<String> lista_messaggi,` è possibile accedere a tale struttura soltanto in **mutua esclusione.** Una volta terminate le istruzioni sopra descritte si entra in un terzo ciclo dal quale è possibile uscire soltanto terminando la fase di editing o se cade la connessione verso il server. Il comportamento è uguale a quello dei due cicli più esterni: si legge l'input e si esegue lo switch dell'operazione da effettuare: `case "end-edit"; case "send"; case "receive";` le operazioni svolte sono rispettivamente: fine della fase di editing e invio del documento al server (*è necessario prima aver chiuso l'editor e salvato il file*); invio di un messaggio sulla chat; stampa a video di tutti i messaggi non letti e rimozione di questi.

Quando il client viene chiuso con il comando **termina** vengono eliminati i file creati per l'editing che erano stati salvati nella directory `"C:\\Client"+Nome_del_cliente.`

In sintesi: il client è costituito tra 3 cicli WHILE annidati, all'inizio di ogni ciclo si legge l'input dell'utente e si effettua una switch per capire di che richiesta si tratti per poterla, eventualmente, inoltrare al server. Uno "schema" semplificativo è il seguente:

```
while(non ricevo il comando di terminazione del client) {
            //leggo input
            switch (richiesta)
                  //vari casi tra cui:
                  case login:
                        while (!logout)
                              //leggo input
                              switch(richiesta)
```

```
                        //vari casi tra cui:
                        case edit:
                              while (!end-edit)
                                    //leggo input
                                    switch (richiesta)
                                    //vari casi
}
```

b.ClientMethodsTuring

Contiene un insieme di metodi usufruiti dal Client per inviare richieste al server e per ricevere le risposte. È stato implementato con lo scopo di "snellire" il codice del client e rendere più leggibile il progetto. A tutti i metodi vanno passati almeno tre parametri: Operazione, socketchannel (per la comunicazione con il server) e bytebuffer (per leggere/scrivere dal/sul canale). Tra i metodi troviamo:

- `public static int sendOp(String Operazione, SocketChannel socketclient, ByteBuffer buffer);`

  Usato per inviare operazioni che richiedono come risposta un semplice flag, ad esempio l'operazione di login o di create.

- `public static int requestedit(String Nome, String Documento, String sezione, SocketChannel socketclient, ByteBuffer buffer, String editor);`

  Usato per richiedere l'editing di una specifica sezione. Dopo aver inviato la richiesta mi metto in attesa di ricevere la dimensione del file che viene espressa come long con valore sempre maggiore o uguale a 1, la ricezione di un valore uguale a 0 mi segnala che l'editing della sezione richiesta non può essere effettuato i motivi possono essere diversi: qualcuno sta già effettuando l'editing della sezione oppure la sezione non esiste. Se l'editing mi viene concesso creo una directory `"C:\\Client"+Nome_del_client` dove inserire tutti i documenti sui ho chiesto e ottenuto il permetto di editing, in modo tale che quando termino invio la copia modificata e salvata nella directory `"C:\\Client"+Nome_del_client.` Infine, avvio l'editor.

- `public static String getIPChat(String Operazione, SocketChannel socketclient, ByteBuffer buffer)`

  Usato per la richiesta di indirizzo e porta della chat, questi vengono interpretati come una stringa da cui, successivamente, il client estrapola i valori.

- `public static boolean getlist(String Operazione, SocketChannel socketclient, ByteBuffer buffer)`

  Usato per la richiesta di show che di list. In entrambi i casi dopo aver inviato l'operazione al server si va a leggere dal canale fin quando non riceviamo un messaggio speciale dal server (rappresentato dalla stringa "finish"), i byte ricevuti vengono interpretati come char e da questi si costruisce una stringa per riga (fino a \n) e le si stampa a video.

- `public static boolean sendSezione(String Operazione, String PathDocumento,  SocketChannel socketclient, ByteBuffer buffer)`

  Usato per inviare il contenuto della sezione modificata. Il metodo invia dapprima la dimensione del file (operazione necessaria perché la dimensione con molta probabilità è cambiata) e successivamente il file avvalendosi dei **FileChannel.**

c. Notify

Implementa l'interfaccia Runnable. Apre una connessione TCP con il server, invia un'unica richiesta a quest'ultimo per comunicargli che si tratta di un Thread di notifica collegato a uno ed uno solo utente. Tutte le notifiche di **share** vengono inoltrate sulla connessione stabilita da tale thread, il suo unico scopo infatti è mettersi in attesa da messaggi del server per poi stamparli a video.

d. ThreadChat

Si occupa di ricevere i messaggi inviati da altri utenti sulla chat relativa alla sezione che si sta editando. L'unica cosa che fa il thread è ricevere i messaggi e salvarli nella struttura dati condivisa `ArrayList<String> lista_messaggi` facendo attenzione ad agire in mutua esclusione.

3.SERVER

a.MainClassTuringServer.

È possibile passare fino a due parametri nel seguente ordine: numero di porta del registro, numero di porta per il socket. Crea la directory in cui verranno salvati i documenti dagli utenti `"C:\\DOCUMENTI-TURING"` e vengono allocate le principali strutture dati:

- `HashMap<String,UtenteOnline> clienti_online` una hashmap che associa univocamente, il nome del cliente che ha effettuato con successo l'operazione di login con un'istanza dell'oggetto UtenteOnline che contiene tutti i dati necessari a supportare le operazioni richieste dall'utente. Quando un utente effettua il logout oppure si chiude bruscamente, l'associazione <Nome, UtenteOnline> viene rimossa dalla tabella hash.

- `HashMap<String, Documento> tabella_documenti` tabella che associa univocamente, un nome di un documento con l'istanza di Documento corrispettiva. Ogni documento allocato, indipendentemente dall'utente che ne richiede la creazione, deve avere un nome univoco, non possono esistere documenti con lo stesso nome.

- `HashMap<String, chat> tab_chat` È una tabella che tiene traccia di indirizzo e porta associati ad ogni documento di cui si è chiesto almeno una volta l'editing.

Segue la creazione di un oggetto remoto e la pubblicazione del riferimento a questo. I client ottenendo il riferimento all'oggetto remoto avranno la possibilità di registrarsi. Si entra, a questo punto, nel ciclo principale del server: ad ogni iterazione vengono selezionate e iterate le SelectionKey pronte in lettura o scrittura.

Ad ogni Selectionkey è associato un attachment UtenteOnline. Se un utente effettua il login con successo l'attachment sarà lo stesso contenuto nella tabella degli utenti online. L'attachment viene sempre aggiunto anche se l'utente non è online, questo è necessario perché l'oggetto di tipo UtenteOnline ci fornisce le indicazioni necessarie per individuare la risposta da inviare al client, tale istanza risulta, quindi, indispensabile per comunicare con il client.

Se il channel è pronto per la lettura, la prima istruzione da eseguire è la lettura dei bytes sul canale e individuare l'operazione che è stata richiesta dal client mediante

`ServerMethods.LeggiOperazione(…) (riga 98)`

otteniamo un array `String []Op` (come fatto con il client), i cui argomenti sono:

<tipo richiesta> <param1><param2>

In questo ordine, inoltre **<param1><param2>** possono non essere presenti (dipende dal tipo di operazione richiesta). Si effettua lo switch del **<tipo richiesta>** il cui *insieme di valori è uguale a quelli definiti per il client con l'aggiunta di due casi:* {Notify, chat}. Osserviamo il comportamento del server a seconda dei casi:

Key.isReadable

- `case "login":` si effettuano un paio di controlli per verificare che l'utente sia registrato e non sia già online, se le condizioni vengono soddisfatte si inserire la coppia <NomeUtente, istanza_UtenteOnline> settando gli opportuni parametri. In caso di fallimento, notare che viene comunque associata un'istanza di UtenteOnline alla SelectionKey (riga 116), ciò è necessario per comunicare al client (quando il canale è pronto per la scrittura) l'esito dell'operazione.

- `case "notify":` È inviata dal thread che si occupa di gestire le notifiche associate a uno specifico utente online, il server salva il socket nell'istanza UtenteOnline del cliente specificato.

- `case "chat":` il client fa sapere che è pronto a ricevere l'indirizzo della chat a cui deve aderire.

- `case "create":` Controlla l'univocità del nome, in tabella_documenti (definita sopra), se soddisfatta si procede alla creazione del documento e all'aggiunta di quest'ultimo nella tabella_documenti. Infine, viene inviato l'esito dell'operazione.

- `case "list":` Si prepara la lista dei documenti da inviare attraverso il metodo:

  ```
  String lista = ServerMethods.getList(…)
  ```

- `case "share":` si effettuano controlli sia sull'utente che ha chiesto la share che sull'utente con cui si vuole condividere il documento (si verifica se è registrato), una volta soddisfatti i requisiti si invia il messaggio di notifica (se online) oppure si salva il messaggio e si aggiunge il nuovo utente nella lista dei collaboratori di quel documento.

- `case "show":` Si verifica se è stato richiesto tutto il documento o una sezione specifica e conserviamo la richiesta all'interno dell'istanza UtenteOnline corrispondente pe completarla non appena il canale è pronto per la lettura.

- `case "edit":` Si verifica che il documento esiste e che l'utente abbia il permesso di modificarla, a questo punto si effettua la "lock" sulla sezione. Il documento verrà spedito in un secondo momento, non appena il canale è pronto per la scrittura. La lock viene eseguita prima di inviare la sezione per "preservarla" all'utente che l'ha chiesta per primo.

- `case "end-edit":` Ci prepariamo a ricevere la sezione.

Key.isWritable:

Si va a recuperare l'operazione da completare, per ogni specifico utente, grazie a un metodo (`.getOperation()`)dell'istanza UtenteOnline associato al cliente online. Poiché diversi casi si comportano nello stesso modo sono stati raggruppati per eseguire la stessa porzione di codice (es: riga 209-2012):

- `case "login"/"create"/"share"/"fail":` invia un intero al client che indica l'esito dell'operazione.

- `case "list"/"msg":` invia una stringa che rappresenta la lista dall'utente oppure un messaggio il cui contenuto varia a seconda dei casi.

- `case "show":` Si vanno a reperire le informazioni relative al completamento della richiesta del cliente grazie all'istanza UtenteOnline associata, si verifica l'esistenza del documento e si invia il contenuto tramite

  ```
  ServerMethods.sendDocumento(…)
  oppure
  ServerMethods.sendShow(…)
  ```

- `case "edit":` si reperisce il nome del documento e si invia la sezione, se ciò avviene con successo si procede andando a estrarre l'indirizzo associato alla chat dalla tabella tab_chat, se presente, altrimenti si invoca

  ```
  ServerMethods.createchat(tab_chat)
  ```

  che fornisce un indirizzo multicast e lo associa al documento. Dopo di che si salva l'indirizzo in un'apposita variabile dell'oggetto UtenteOnline associato al cliente e si attende che il client richieda l'indirizzo della chat mediante il <tipo di richiesta = chat>, a quel punto può essere inviato l'indirizzo. Il motivo per cui non viene inviato subito è che il cliente deve prima leggere la

sezione ricevuta e salvarla in un apposito file e solo successivamente può passare alla ricezione dell'indirizzo di chat.

- `case "end-edit"`:I primi bytes ricevuti riguardano la dimensione della sezione, una volta letti i primi 8 bytes, questi vengono tradotti in un long che chiamiamo "size" e si procede alla lettura del canale fino a quando non leggo al più "size" bytes. Eseguo l'unlock della sezione.

b.ServerMethods

Insieme di metodi utilizzati per ricevere/inviare dati dal/verso i client, ma contiene anche altri metodi utili al server. È stato implementato per semplificare e snellire il codice. Tra i metodi troviamo

- `public static chat createchat(HashMap<String,chat> tab_chat):` quel che si fa è cercare un indirizzo multicast Random da poter assegnare al documento. Dopo aver creato un indirizzo random si eseguono un paio di controlli: dapprima si verifica che l'indirizzo generato non appartenga alla classe degli indirizzi speciali e in seguito si itera tutta la tab_chat (contenente tutti gli indirizzi assegnati ai vari documenti) per essere sicuri di non aver già assegnato l'indirizzo generato ad un altro documento. Se queste condizioni sono soddisfatte si procede cercando una porta libera. Restituisce un oggetto di tipo chat da assegnare al documento.

- `public static int sendShow(…)`: ed `public static int sendDocumento(…)` vengono usati per la show. Il primo invia una singola sezione mentre il secondo esegue un ciclo for dove invia una sezione alla volta invocando il metodo **sendShow**()

- `static void disconnect` sempre invocato quando un client si disconnette per vari motivi, si occupa di chiudere il canale, il socket del thread notify e se stava effettuando l'editing di una sezione, questa viene unlocked. L'utente viene infine rimosso dalla tabella dei clienti online.

4. INTERFACCE REMOTE E LE LORO IMPLEMENTAZIONI

a. Registrazione e RegistrazioneImpl

RegistrazioneImpl estende *UnicastRemoteObject* e implementa l'interfaccia *Registrazione.* Viene istanziata e pubblicata dal server. il suo unico attributo è una tabella hash che associa univocamente un nome di un cliente a un'istanza di User. Il client utilizza soltanto il metodo:

```
public synchronized boolean registrami (String Name, String Password)
```

per richiedere di essere registrato al servizio offerto dal server, è synchronized perché più client possono cercare di registrarsi con lo stesso username nello stesso momento e per evitare i conseguenti problemi si effettua la registrazione soltanto in mutua esclusione. il server, invece, utilizza il metodo

```
public synchronized User is_register (String Name)
```

per verificare, quando opportuno che un certo cliente è registrato prima di poter eseguire le sue richieste.

b. UserInt e User

User Estende *UnicastRemoteObject* e implementa *Userint*, un'istanza di questa classe contiene le informazioni "permanenti" di un utente registrato, ovvero, contiene quegli attributi che vengono comunque "conservati" (e anche modificati) indipendentemente dal fatto che il cliente sia online oppure no. Un'istanza viene associata ad uno ed uno solo utente e viene creata quando questo effettua l'operazione di registrazione: viene definita un'associazione tra nome utente e istanza User e viene inserita nella tabella hash di *Registrazione*. **La stessa istanza**, viene anche mantenuta tra gli attributi dell'oggetto UtenteOnline quando un cliente effettua il login.
Tra gli attributi di User, troviamo:

- `protected ArrayList<String> lista_documenti:` Lista contenente tutti i documenti su cui può effettuare l'editing, comprende sia quelli creati che quelli ottenuti tramite invito.
- `protected Vector<String> messaggi_pendenti:` Lista dei messaggi di notifica di sharing, quando l'utente non è online i messaggi vengono conservati in tale struttura per poi essere inviati e rimossi quando l'utente effettua il login.
- `protected String Password:` Password associata all'utente

5.ALTRE CLASSI

a.chat

tiene traccia della coppia <indirizzo, numero di porta> che viene assegnata alla chat di un documento. Un'istanza di questo tipo viene inserita nella **tab_chat** (definita nel server) con chiave <NomeDocumento>. È necessario tenere traccia di tutte le istanze **chat** per garantire l'univocità degli indirizzi multicast assegnati.

b.Documento

Il documento "vero e proprio" è una directory e le sue sezioni dei file ".txt".
Un'istanza Documento, contiene tutti gli attributi e metodi necessari per la gestione del documento da parte del server (non viene usato dal client). Tra gli attributi, troviamo: il path del documento, il nome del documento, il nome del creatore del documento, il numero di sezioni da cui è composto (questi attributi una volta stabiliti non possono essere più modificati), il numero di utenti che sta effettuando l'editing sul documento, l'insieme degli utenti che possono modificare il documento e una struttura che indica quali sezioni sono in fase di editing. Tutti i documenti vengono salvati nella directory: `"C:\\DOCUMENTI-TURING\\"` Il path è dato da `"C:\\DOCUMENTI-TURING\\"+NomeDocumento` mentre le sezioni saranno salvate con il Nome "`sezione"+Numero_sezione +".txt".` La classe fornisce, tra vari metodi, la lock e unlock delle sezioni e possibilità di aggiungere nuovi collaboratori al documento.

c.UtenteOnline

Un’istanza di questa classe viene associata ad ogni cliente che effettua il login (avremo un’associazione nella tabella dei clienti online con il nome dell’utente), lo stesso riferimento lo si trova anche nell’attachment della SelectionKey associata a un client che stabilisce una connessione tcp con il server, in questo caso è possibile che una SelectionKey abbia associato tale istanza anche se il cliente non è online, questo perché un oggetto UtenteOnline contiene degli attributi necessari per inviare la corretta risposta al client.
Ad esempio: un client richiede l’operazione di login, ma questa non va a buon fine. È necessario inviare un messaggio al client per comunicargli che il login è fallito, tale messaggio però va spedito in un secondo momento (quando il canale è pronto per la scrittura) e quindi è necessario salvare la risposta da qualche parte e per farlo si utilizza un’istanza UtenteOnline. Gli attributi formano lo stato attuale del cliente, e tra questi troviamo: un buffer, il tipo di operazione da completare, il socket per le notifiche e quello per la “comunicazione standard”, username, il riferimento all’oggetto User associato al cliente, il documento e la sezione di cui si è chiesto edit o la show, un boolean che indica se il cliente sta effettuando l’editing e una stringa che rappresenta un messaggio da inviare al client.