Progetto di Fine Corso

A.A. 2818/19

di Laboratorio di Reti, Corsi A e B

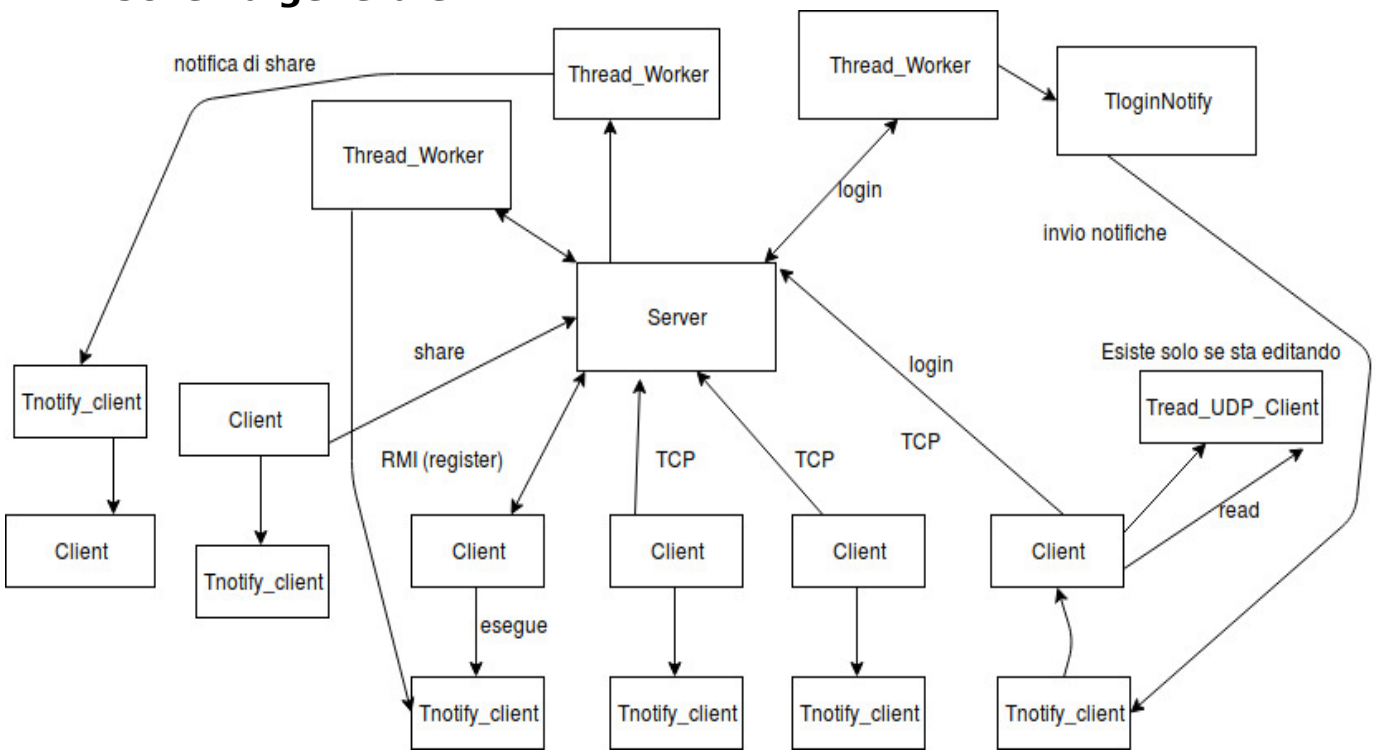
---

disTribUted collaboRative edItiNG

1. Introduzione

disTribUted collaboRative edItiNG è un programma applicativo, che può essere utilizzato da più Client in contemporanea, in grado di modificare documenti di testo. TURING permette di creare, modificare e mostrare file di testo attraverso una "*command-line interface*". Per utilizzare tale programma sono messi a disposizione alcuni comandi (Vedi Sez. 5) che aiuteranno il Client a interagire in maniera semplice con esso. L'implementazione del progetto prevede che alla creazione di un nuovo documento, il creatore è identificato come "master" mentre un collaboratore come "contributor".
Si usano le seguenti notazioni: metodo, *istruzioni o esempi.*

Schema generale



2. Classi

2.1. Client.java

2.1.1. Main. Il Main del Client è strutturato in modo tale che nella prima parte del codice sono presenti tutte le strutture dati utilizzate durante lo svolgimento del progetto.
Il Client viene eseguito attraverso il seguente comando: java Client e i seguenti argomenti:

1. Indirizzo IP remoto o locale del Server per eseguire la lookup ed ottenere l'oggetto remoto "Server" (Vedi Sez. 5).
2. Porta per identificare il servizio identificato dalla porta "port".
3. Indirizzo IP remoto o locale per instaurare le connessioni TCP ed eseguire le richieste del Client
4. Porta che identifica il processo durante le connessioni TCP
5. Porta che identifica il processo di ricezione per le notifiche di invito da parte di uno User *U1* uno User *U2* ad editare un documento creato dallo User *U1*.

2.1.2. Switch comandi. Il cuore del Client è lo switch che stiamo per descrivere. Una volta che l'utente tramite tastiera inserisce un comando con i relativi argomenti, lo switch permette di gestire le varie operazioni richieste dal progetto. Lo switch è stato progetto per seguire l'automa FSM pubblicato nel testo del progetto; attraverso due variabili booleane "logged" e "editing" riusciamo a rappresentire gli stati richiesti dal progetto, diminuendo così il numero di connessioni TCP;

2.1.3. setCommunication. Instaura una connessione TCP ogni volta che viene richiamato questo metodo all'indirizzo IP e la porta specificata.
Una volta che viene ottenuto il socket, vengono configurati gli stream socket per poter comunicare con il Server.

2.1.4. readFromServer. Dopo aver aperto una connessione TCP col Server attraverso il metodo sopracitato questo metodo consente di leggere dati dal Server al Client leggendo riga per riga.

2.2. Server.java. Come per il Client anche nella prima parte troviamo tutte le variabili necessarie per l'esecuzione del Server; per poterlo eseguire si utilizzano gli stessi argomenti del Client (tranne per il quinto che non è necessario). Il Server è realizzato in multithread, ciò significa che ogni richiesta del Client viene gestita in un thread separato. Come per il Client anche il cuore del Server è uno switch che interpreta i comandi ricevuti tramite connessioni TCP da parte dei vari Client.

2.3. ServerInterfaceImpl.java. Questa classe implementa l'interfaccia ServerInterface per poter ricevere richieste di registrazione da parte dei Client sfruttando il servizio RMI. Inoltre tiene traccia di tutti gli utenti registrati in modo tale da non accedere sempre al file degli utenti registrati su disco. La registerUser permette di registrare un Utente; una volta che l'utente è stato registrato viene notificato attraverso il metodo sendMsg.

2.4. ClientInterfaceImpl.java Questa classe implementa l'interfaccia per la gestione delle credenziali di un utente attraverso i metodi: getName, getPassword e checkPermission. Tramite il metodo sendMessage il Client riceve messaggi dal Server sfruttando il servizio RMI.

2.5. InterfaceClient.java. Interfaccia della classe descritta alla sezione 2.4

2.6. ServerInterface.java. Interfaccia della classe descritta in 2.3

2.7. UserOnline.java. Questa classe tiene traccia di tutti gli utenti che hanno effettuato la login tramite il metodo put; possiamo ottenere informazioni sugli indirizzi IP dei vari Client tramite il metodo getAddress o disconnettere un

Client tramite il metodo remove. Gli altri metodi presenti sono utilizzati per tenere lo stato in modo consistente della classe.

2.8. UserEditing.java. Questa classe viene utilizzata per tenere traccia degli utenti che stanno editando una sezione del documento. Essendo tale classe condivisa tra più thread è gestita in mutua esclusione. I metodi implementati in questa classe sono: editDocument per registrare un nuovo utente che sta editando, removedDocument per rimuovere un utente che ha terminato di editare. Infine il metodo checkDocumentEditing permette di verificare che un utente stia editando una particolare sezione di un documento.

2.9. OfflineMessage.java. Questa classe permette di tenere memorizzati tutti gli inviti da parte di un generico utente U1 ad un altro generico utente U2 quando quest'ultimo non ha effettuato la login. Una volta che l'utente U2 si collega,dunque effettua la login, tramite le classi descritte nelle sezione 2.10 e 2.11 riceve una notifica di tutti gli inviti ricevuti da parte dei master dei vari documenti.

2.10. TloginNotify.java Questa classe implementata come un thread viene eseguita al momento della login di un nuovo utente come supporto al Server. Ottiene l'indirizzo ip e la porta, per poi aprire una nuova connessione TCP (verso il Client) notificando l'utente con gli inviti da parte degli altri utenti, se presenti.

2.11. Tnotify_Client.java. Questa classe implementata come un thread, viene eseguita quando un utente effettua la login come supporto al Client, ricevendo eventuali notifiche di invito da parte di altri utenti per l'editing del documento.

2.12. MessageReceivedUDP.java. Questa classe contiene tutti i messaggi inviati da altri utenti (o da me stesso) durante l'editing di un documento. Tramite il metodo insertMessage l'utente memorizza i messaggi che vengono inviati in multicast a tutti gli utenti che fanno parte di uno stesso gruppo (ovvero stanno editando uno stesso documento). Tramite il metodo getMessage l'utente legge i messaggi ricevuti durante l'editing. Poichè l'utente può leggere o inviare i messaggi solo durante la fase di editing di un documento il metodo deleteAllMessage riporta lo stato iniziale di tale classe quando un utente termina di editare un documento.

2.13. Tread_UDP_Client.java. Questa classe implementata come un thread permette di ricevere i messaggi inviati, tramite il protocollo multicast UDP, dall'utente stesso o dagli altri utenti. Ogni volta che il thread riceve un nuovo messaggio lo inserisce nella struttura dati presente nella classe definita nella sezione 2.2.12.

2.14. Group_Multicast_UDP.java. Questa classe tiene traccia di tutti gli indirizzi multicast associati ai vari documenti. Ogni qual volta un utente edita una sezione del documento e questo non era editato da nessun altro tramite il metodo createNewGroup viene associato un nuovo indirizzo multicast e creato un nuovo gruppo. Una volta che l'ultimo utente rimasto ad editare il documento termina l'editing viene invocato il metodo removeGroup per dissociare l'indirizzo dal gruppo.

2.15. Tworker_Server.java. Il cuore di questo progetto è questa classe. Quest'ultima viene eseguita come un thread separato ogni volta che l'utente effettua una nuova richiesta al Server. Come per Client.java e Server.java uno switch permette di gestire tutte le operazioni richieste dal progetto. Questo thread si occupa di fornire tutti messaggi di risposta positivi, negativi o relativi ai permessi alle richieste del Client. Questa classe necissita delle stesse strutture dati presenti nel Server.java, e può essere vista come un'implementazione d'interfaccia relativa a Server.java

3. Strutture Dati

Per poter realizzare il servizio applicativo TURING, l'implementazione del progetto è stata incentrata principalmente su l'uso di HashMap<String,String> in grado di memorizzare tutte le informazioni necessarie per una corretta esecuzione del programma applicativo.

1. **Group_Multicast_UDP.** Questa classe, per scelta implementativa, utilizza una HashMap<String,String> la quale memorizza una coppia (chiave,valore) che identifica (nome del gruppo, indirizzo ip multicast). Poichè è utilizzata da più thread per poter creare o rimuovere gruppi multicast, una Lock permette di gestire l'unica sezione critica che questa classe ha (ovvero l'HashMap).
2. **MessageReceivedUDP.** Questa classe, per scelta implementativa, utilizza una StringBuilder per poter costruire l'intera chat ottenuta ricevendo messaggi multicast UDP. Poichè è condivisa da due thread (Il Client che esegue la read dei messaggi e Tread_UDP_Client che fonirsce a questa classe i messaggi ricevuti in multicast) anch'essa ha una Lock per poter gestire l'unica sezione critica che questa classe ha (ovvero la StringBuilder).
3. **OfflineMessage.** Questa classe, per scelta implementativa, utilizza una HashMap<String,String> la quale memorizza una coppia (chiave,valore) che identifica (l'username al quale inviare il messaggio, lista di documenti da editare). Poichè questa struttura è condivisa da più thread, una Lock permette di gestire l'unica sezione critica che questa classe ha (ovvero l'HashMap).
4. **Server**. Come menzionato nella sezione 2.2, il Server è realizzato in multithreading. Ciò implica la necessità di avere un ThreadPoolExecutor che gestica tutti i thread che lavorano in contemporanea. Per poter realizzare l'intero progetto il Server mantiene le strutture dati principali (UserEditing, UserOnline,OfflineMessage,Group_Multicast_UDP) al suo interno; ad ogni richiesta da parte del Client tali strutture vengono fornite a un thread separato. Quest'ultimo si occuperà di realizzare la richiesta effettuata dal Client.
5. **ServerInterfaceImpl.** Questa classe per poter mantere traccia di tutti gli utenti registrati, utilizza una HashMap<String,String> la quale memorizza una coppia (chiave, valore) che identifica (username,password). Dunque tale struttura permette di effettuare la registrazione di tutti i Client. Essendo un servizio RMI, tutti i metodi implementati sono synchronized dunque non abbiamo bisogno di nessun supporto esplicito per gestire la mutua esclusione sulla struttura dati.
6. **TloginNotify.** Questa classe non ha strutture dati personali. Utilizza dunque strutture dati di supporto definite in altre classi per realizzare il proprio compito. Tramite l'istanza della classe OfflineMessage (Sez 2.9) vengono recuperate tutte le notifiche, indirizzate ad un Utente U1, al momento della login, e spedite al Client dopo aver ottenuto l'indirizzo IP e la porta per comunicare tramite l'istanza della classe UsersOnline (Sez 2.7)
7. **Tread_UDP_Client.** Questa classe non ha strutture dati personali. Utilizza dunque una struttura dati di supporto per poter memorizzare i messaggi ricevuti in multicast. Ogni qual volta viene ricevuto un messaggio, tramite l'istanza della classe MessageReceiveUDP (Sez. 2.2.12 e Sez. 3.2) si tiene traccia di tutta la chat relativa ad un determinato gruppo.

8. **UserEditing.** Questa classe per poter mantere traccia degli utenti che stanno editando le varie sezioni dei documenti, utilizza una HashMap<String,String> la quale memorizza una coppia (chiave, valore) che identifica (utente, sezione che sta modificando). Come richiesto da progetto, un gruppo multicast viene rimosso quando nessun utente sta editando più un particolare documento, dunque questa classe fornisce supporto alla classe Group_Multicast_UDP (Sez. 3.1 e 2.2.14) per creare o rimuovere un gruppo e il relativo indirizzo IP multicast. Poichè l'istanza di questa classe definita nella classe Server.java è condivisa tra più thread, una Lock permette di garantire la mutua esclusione.
9. **UserOnline.** Questa classe per poter mantere traccia degli utenti "loggati", utilizza una HashMap<String,String> la quale memorizza una coppia (chiave, valore) che identifica (username, indirizzo_ip:porta). La porta viene utilizzata per ricevere notifiche al momento della login tramite le due classi TloginNotify (Sez. 2.10) e Tnotify_Client (Sez 2.11). Poichè questa struttura dati è condivisa tra più thread, quest'ultima necessita di una Lock per poter gestire la mutua esclusione.

4. Directory

Per veriricare il corretto funzionamento del progetto, per scelta implementative sono state create alcune directory di supporto alla gestione dell'intero sistema.

1. **Files.** In questa directory sono contenuti tutti i documenti e le relative sezioni. Una volta che un utente U1 decide di creare un nuovo documento "doc" con "n sezioni" all'interno di files/doc/ troveremo tutte le sezioni nominate da sez_0.txt a sez_n.txt.
2. **Editing.** In questa directory sono contenute delle sottodirectory (i documenti che vengono editati) al cui interno vi sono le sezioni che i vari utenti stanno editando. Dunque se un utente "U1" decide di editare la "sezione 0" del documento "doc", all'iterno della directory editing/doc troveremo un file chiamato sez_0.txt che potrà essere modificato. Una volta che l'utente U1 ha terminato l'editing, il file files/doc/sez_0.txt verrà sovrascritto con il file editing/doc/sez_0.txt e quest'ultimo viene cancellato.
3. **Users.** Questa directory contiene un unico file "registered.txt" contenente tutti gli utenti registrati (uno per riga) e la relativa password. Al momento dell'esecuzione del Server, la classe UserOnline (Sez. 2.2.7 e Sez. 3.9) carica tutti i dati presenti in questo file in modo da tener traccia di tutti gli utenti registrati anche quando il Server esegue una shutdown.

5. Comandi

Nessun comando è stato aggiunto oltre quelli richiesti dal progetto.

1. **Esecuzione del Server**

   java Server localhost port localhost port2

   oppure *java Server ip port ip port2*

2. **Esecuzione del Client (port3 necessaria per le notifiche offline)**

   java Client localhost port localhost port2 port3

oppure *java Client ip port ip port port3*

3. **Comandi di interazione**

   Ciascuna richiesta può essere eseguita se e soltanto se è consona allo stato in cui si trova.

   Istruzioni. *turing --help*

   Passo 1. Registrazione utente

   turing register username password

   Passo 2. Login utente

   turing login username password

   Passo 3. Comandi eseguibili:

   1. **Creazione documento**

      È possibile creare un nuovo documento (se non esiste già) tramite:

      turing create nome_documento numero_sezioni

   2. **Mostrare il documento**

      È possibile vedere il documento (se esiste) tramite:

      turing show nome_documento

   3. **Mostrare una sezione del documento**

      È possibile vedere una sezione del documento (se esiste) tramite:

      turing show nome_documento num_sez

   4. **Editare un documento**

      È possibile editare una sezione del documento (se esiste) tramite:

      turing edit nome_doc num_sez

   5. **Terminare l'edit di un documento**

      È possibile terminare l'edit di una sezione (se la si stava già editando) tramite:

      turing end – edit nome_doc num_sez

   6. **Inviare messaggi**

      È possibile inviare messaggi (se si sta editando un documento) tramite:

      turing send messaggio (Attenzione gli spazi non sono ammessi)

   7. **Ricevere messaggi**

      È possibile ricevere messaggi (se si sta editando un documento) tramite:

      turing read (Attenzione gli spazi non sono ammessi)

   8. **Condividere un documento**

È possibile condividere un documento (se non stiamo editando) tramite:

turing share nome_doc nome_user (Attenzione l'utente deve essere registrato altrimenti non è possibile condividere il documento).

9. **Ottenere la lista dei documenti**

   È possibile ottenere la lista dei documenti tramite:

   turing list

10. **Logout**

    È possibile effettuare la logout (se non stiamo editando alcun documento) tramite:

    turing logout.

Esempio di utilizzo:

Client 1	*Client2*
turing register a b	*turing register b d*
turing login a b	*turing login b d*
turing create doc 7	*turing edit doc 0 (non può)*
share doc b (dopo aver registrato b)	*turing edit doc 1*
turing edit doc 0	*turing send ciao*
turing send ciao	*turing read*
turing read	*turing end – edit doc 0 (non può)*
turing end – edit doc 0	*turing end – edit doc 1*
turing logout	*turing logout*

7. Osservazioni

Per l'implementazione del progetto sono stati riadattati alcuni codici a tre cifre TCP per poter comunicare più efficientemente.

1. **Richiesta completata correttamente**
   1. **200.** Richiesta eseguita con successo
   2. **201.** Finito di ricevere una sezione correttamente
   3. **202.** Finito di ricevere la sezione o l'ultima sezione dell'intero documento correttamente.
2. **Permessi negati**
   1. **301.** Permessi non sufficienti ad eseguire l'operazione richiesta
   2. **302.** Utente inesistente
3. **Richiesta non completata correttamente**

1. **300.** Il file richiesto non esiste
2. **303.** Non esiste alcun documento, o la directoy non esiste
3. **304.** L’utente sta già editando (mai restituito questo errore)
4. **305.** La sezione non può essere editata (lo sta facendo qualcun altro)
2. **306.** L’utente non sta editando alcun file
3. **307.** L’utente sta editando una sezione differente

Davide Montagno Bozzone