Relazione Progetto WORTH

Architettura del sistema

Il sistema è suddiviso in 2 macro-entità: il client e il server.
Il client ha il compito di leggere le richieste dell'utente e di inoltrarle al server, per poi aspettare la risposta e stamparla come output all'utente; mentre il server ha il compito di ricevere le richieste, di eseguirle, se possibile, e di mandare un feedback testuale all'utente. Come da specifica, la comunicazione tra client e server è implementata attraverso una connessione TCP, che viene sfruttata per tutti i comandi che hanno bisogno dell'intervento del server per essere eseguiti, tranne per la registrazione, dove viene usato RMI. Per le operazioni riguardanti la chat, invece, vengono usati dei thread (tanti quanti il numero di progetti di cui è membro l'utente) che ricevono i messaggi inoltrati sul gruppo multicast del progetto e li salvano su una struttura dati.

Per prima cosa il client ottiene un riferimento al server tramite una lookup al registro remoto creato dal server. Dopodichè instaura ed apre la connessione TCP. A questo punto il client è in grado di ricevere richieste dall'utente. Quando la richiesta ha bisogno dell'intervento del server per essere soddisfatta, viene invocato il metodo *communicateWithServer* che invia la richiesta, aggiungendo, quando necessario, il nome dell'utente come parametro, in modo tale che per il server sia più facile conoscere l'username dell'utente che l'ha effettuata. Una volta eseguita l'operazione il metodo restituisce la risposta del server. Nel caso particolare della registrazione, invece, viene invocato il metodo remoto *register*, tramite il riferimento al server ottenuto in precedenza. Nel caso del login, dopo aver ricevuto la risposta del server, si controlla se l'operazione è andata a buon fine, in tal caso il client si registra al servizio di callback per l'aggiornamento sugli utenti e sui progetti, al quale viene aggiunto passando come parametro uno stub che lo identifica. In risposta all'operazione di iscrizione alle callback il client riceve la lista di tutti gli utenti registrati con il loro stato (online/offline) e la lista dei progetti di cui fa parte; dopo la ricezione di quest'ultima, vengono messi in esecuzione dei thread, tanti quanti sono i progetti della lista, per la ricezione dei messaggi. Viceversa, per il logout, il client si disiscriverà dal servizio di callback e terminerà tutti i thread precedentemente attivati. L'operazione di lettura della chat viene effettuata con il metodo *getMessaggi*, che interroga il task delegato alla chat del progetto e restituisce tutti i messaggi non ancora letti dall'utente. Per poter inviare un messaggio alla chat di progetto, il client manda un pacchetto UDP al gruppo multicast del progetto, il cui contenuto è il messaggio scritto dall'utente concatenato all'username, in modo tale che leggendo la chat si capisca chi ha inviato i messaggi.

Per avere buone prestazioni e scalabilità il server è implementato usando il multiplexed I/O con NIO; il comportamento delle operazioni di I/O è settato a non bloccante e la scelta delle operazioni da fare è gestita per mezzo di Selettori. Per

prima cosa il server recupera tutti i dati su utenti, progetti, membri e cards. Il ripristino delle informazioni viene effettuato andando a leggere i file di backup contenuti nella cartella del server. Per quanto riguarda utenti e cards è stato necessario persistere l'intero oggetto, per questo si è optato per la serializzazione tramite json; per quanto riguarda i membri del progetto e la storia delle card, invece, è stato necessario salvare solo delle stringhe (username per i membri, nome degli stati per le card), perciò si è optato per il salvataggio dei dati su file txt. Una volta ottenuti tutti i dati il server apre la connessione TCP, setta il comportamento non bloccante, crea il Selector e si prepara a ricevere richieste. Dopo aver instaurato una connessione con un client, il server attende le sue richieste. Quando ne arriva una, viene suddivisa in comando effettivo e parametri, dopodiché viene invocato il metodo che esegue l'operazione, il quale creerà anche la risposta di riscontro da inviare al client. Quando il server riceve il comando *quit*, esso provvederà a chiudere la connessione col client, ma non prima di aver inviato la conferma, in modo tale che anche il client possa chiudere la connessione senza effetti indesiderati. Il server prevede anche un meccanismo per gestire eventuali crash improvvisi lato client, grazie ad una struttura dati che associa la socket all'username dell'utente che la sta sfruttando. Il server ricava l'username dalla struttura dati e in caso fosse necessario esegue il logout al posto suo, dopodichè chiude la connessione. Ogni metodo utilizzato dal server esegue tutti i controlli necessari per verificare se l'utente ha i diritti per effettuare l'operazione o se l'operazione non può effettivamente essere eseguita, inoltre, dove necessario, esegue operazioni di I/O sui file di backup (vedere sopra). Il server dispone anche di metodi remoti che vengono usati per le azioni delegate al RMI, come la registrazione di un utente, l'iscrizione ad una callback, e l'invio di una notifica dell'avvenimento di un certo evento ad uno o più client (callback). Per il funzionamento delle chat di progetto il server assegna ad ognuno un indirizzo multicast diverso, compreso fra 224.0.1.0 e 239.255.255.255, e un numero di porta per la socket (in totale possono esserci, contemporaneamente, 268.435.200 progetti e chat attive). Per ricavare questi valori il client esegue i metodi *getChatMulticastAddress* e *getChatMulticastsocketNumber* sull'istanza del progetto.

Threads e concorrenza

Per quanto riguarda il server, la concorrenza e l'amministrazione dei thread è gestita in modo automatico dal meccanismo di multiplexing I/O con NIO, salvo per le richieste di registrazione, che non sfruttano la connessione TCP. I metodi chiamati da remoto hanno bisogno di essere eseguiti in modo atomico perché accedono concorrentemente alle stesse strutture dati (*clients*, *registeredUsers*), e quindi acquisiscono la lock sulle sezioni critiche in cui usano certe strutture dati. La concorrenza si ha sia fra i metodi remoti, sia fra metodi remoti e metodi usati per le richieste TCP, ecco perché anche il metodo *register* (usato per la creazione di un utente) presenta la dicitura *synchronized*.

Il client è formato da un thread principale, che legge le richieste dell'utente, le invia al server (se necessario) e stampa le risposte; e da altri thread secondari che hanno il compito di salvare i messaggi ricevuti sulle chat di progetto. Il thread principale attiva i thread o al momento del login o quando viene notificato all'utente di essere stato aggiunto ad un progetto; ed esegue la join quando il progetto viene chiuso o dopo aver eseguito il logout. Per poter rendere queste operazioni possibili il client memorizza in due liste i thread e le corrispondenti istanze di *MessageReader*. In questo caso la concorrenza viene gestita in modo tale da evitare conflitti sulle variabili che vengono aggiornate a seguito di una callback (*chatReaders*, *threadReaders* e *mappaStatoUtenti*), poiché queste variabili sono accessibili in altri contesti, indipendenti dalla callback.

All'interno del task per la lettura dei messaggi della chat è necessario gestire la concorrenza sulla lista che contiene i messaggi. Essa infatti può essere modificata sia dal metodo *run()* dopo aver ricevuto un pacchetto UDP, sia dal client che chiede di prelevare tutti i messaggi dalla lista.

Per quanto riguarda i metodi delle altre classi, la gestione della concorrenza non è necessaria perché vengono chiamati da contesti in cui ne è già garantito il corretto funzionamento.

Classi e Interfacce

Le classi e le interfacce del progetto sono le seguenti:

- ***MainClient***: la classe che contiene il metodo main in cui viene richiamato il metodo che mette in esecuzione il client.
- ***MainServer***: la classe che contiene il metodo main in cui viene richiamato il metodo che mette in esecuzione il server.
- ***NotifyEventInterface***: interfaccia che definisce i metodi remoti del client, i quali vengono usati dal server per le callback. Definisce anche il metodo *getUserLogged*, per far sì che il server sappia quale utente è in esecuzione su un determinato client.
- ***Client***: classe che implementa le funzionalità del client. Tra queste abbiamo i metodi usati dal server per le callback, i metodi per risolvere richieste senza inoltrarle al server e i metodi che vengono usati per comunicare con il server. Il metodo principale della classe (*start()*) ha la fondamentale funzione di leggere ed interpretare ciò che scrive l'utente, e in base al comando letto decide cosa fare per soddisfare la richiesta.
- ***MessageReader***: classe che implementa il task delegato a leggere i messaggi inviati su un certo gruppo multicast e a salvarli in una struttura dati al quale il client può accedere quando vuole.

- ***ServerInterface***: interfaccia che definisce i metodi remoti del server, tra i quali c'è il metodo per creare un nuovo account e i metodi per iscriversi o disiscriversi dalle callback.
- ***Server***: classe che implementa le funzionalità del server. Tra i compiti principali della classe ci sono quelli di immagazzinare e cercare in altri file tutte le informazioni utili, di analizzare le richieste del client ed eseguirle, di rispondere al client con un messaggio che indica se l'operazione è andata a buon fine o no e di notificare ai client l'avvenimento di certi eventi. La funzionalità principale (quella di ricevere e rispondere alle richieste) è implementata nel metodo *start()*, il quale a sua volta richiama altri metodi privati che controllano la correttezza e ammissibilità della richiesta per poi, in caso, eseguirla.
- ***Utente***: classe che modella e implementa le funzionalità dell'utente di worth, il quale è identificato da un username e possiede una password privata.
- ***Progetto***: classe che modella e implementa le funzionalità dei progetti in worth, i quali sono identificati da un nome di progetto e posseggono una lista di card per ogni possibile stato, una lista dei membri e delle proprie configurazioni per implementare il gruppo multicast della chat.
- ***Card***: classe che modella e implementa le funzionalità delle card dei progetti, le quali sono identificate da un nome e posseggono una descrizione. All'interno dello stesso progetto non ci possono essere più card con lo stesso nome. Gli stati in cui si possono trovare sono TO DO, IN PROGRESS, TO BE REVISED e DONE.

Scelte progettuali

Di seguito descrivo alcune scelte progettuali particolari spiegandone anche i motivi.

Lettura e scrittura su file

Per la lettura e scrittura dei file di testo vengono usati dei BufferedReader e dei BufferedWriter che eseguono le operazioni di I/O su dei buffer interni e non direttamente sugli stream che comunicano coi file. Questa scelta è data dal fatto che leggere e scrivere su buffer interni è più veloce che leggere byte a byte da uno stream, comportando ad un aumento delle prestazioni. Per quanto riguarda, invece, la scrittura dei file json, viene effettuata con la funzione *writeValue*, il cui risultato, però, viene scritto tra «{"*nome_array*":» e «}» (*nome_array* è uguale a *utenti* e *cards*, rispettivamente per cosa si vuole serializzare). Questo aggiustamento viene fatto in modo tale che la lettura del file json avvenga in modo semplice e veloce, memorizzando su una variabile JsonNode gli oggetti deserializzati grazie alla funzione *objectMapper.readTree(fileReader).get("nome_array")*.

Generazione indirizzi multicast

Gli indirizzi multicast disponibili per il progetto partono dal 224.0.1.0 fino al 239.255.255.255, garantendo un alto numero di chat attive in contemporanea. Per assegnare un indirizzo multicast ad un nuovo progetto si parte dal primo e si controlla se è libero, in tal caso viene assegnato al progetto, altrimenti viene incrementato finché non se ne trova uno libero o finché non si raggiunge il limite. Questo meccanismo, per quanto dispendioso perché ad ogni creazione di un progetto potenzialmente controlla un alto numero di indirizzi, ha lo scopo di rendere ogni indirizzo riutilizzabile non appena un progetto viene chiuso. Per poter sapere se un indirizzo è attualmente in uso, si salvano tutti quelli occupati in una lista chiamata *multicastAddressInUse*. L'informazione riguardante indirizzo multicast e numero di porta per la chat di ogni progetto non viene persistita, infatti il server, durante la fase di backup, assegna un qualsiasi indirizzo multicast libero ai vari progetti che man mano recupera.

Gestione chat (client)

Per l'invio di messaggi e la lettura delle chat, tutte le operazioni sono svolte lato client. Per la lettura viene richiamato un metodo sull'istanza del MessageReader relativo ad un progetto, e si ottiene una lista di stringhe corrispondenti ai messaggi. Per la scrittura il client ricava l'indirizzo multicast e il numero di porta, dopodiché crea un *DatagramPacket*, una *DatagramSocket* e invia il primo, per mezzo della seconda, al gruppo multicast ricavato precedentemente. Per sapere quali siano l'indirizzo e la porta il client non invia una richiesta al server; bensì le ricava direttamente dai dati salvati nello stato del MessageReader relativo al progetto voluto. L'operazione viene fatta grazie al metodo *getProgetto()* del MessageReader e dal metodo *getChatMulticastAddress()* del Progetto. Queste informazioni vengono depositate direttamente dal client in due modi: al momento del login, come risposta alla richiesta di iscrizione alla callback per i nuovi progetti, riceve una lista di tutti i progetti di cui è membro, a quel punto crea i task per la lettura della chat e memorizza il progetto a cui corrisponde nel suo stato. Oppure, quando il server invia una callback per far sapere all'utente che è stato aggiunto ad un progetto, attraverso il metodo *notifyNewProject*. Esso è implementato in modo tale che venga creata immediatamente un nuovo task per la lettura dei messaggi ed un thread che lo metta in esecuzione; *notifyNewProject* prende come parametro il progetto da memorizzare sullo stato del nuovo MessageReader.

Gestione crash utente

Per non incappare in comportamenti indesiderati nel caso in cui una connessione tra client e server venga interrotta improvvisamente, il server esegue una procedura di gestione dei crash lato client. Per questa serie di operazioni viene usata una HashMap *utentePerConnessione* che associa ogni SocketChannel attiva all'username dell'utente che la sta utilizzando. Quando una connessione viene interrotta in modo anomalo, il server controlla se ci fosse un utente associato ad essa, in tal caso setta il suo stato ad offline e lo notifica agli altri utenti; infine elimina

la connessione da *utentePerConnessione*, la chiude e cancella la chiave. Questo meccanismo non è sufficiente poiché rimane un ROC, che si riferisce al client crashato, nella lista *clients*, contente tutti i riferimenti ai client iscritti alle callbacks. Non appena si proverà ad inviare una callback a quel client, verrà sollevata l'eccezione *ConnectException*; a questo punto basterà che il server la catturi quando necessario e, in tal caso, disiscriva dalle callback il client che l'ha causata.

Gestione dei buffer

Per la comunicazione tra socket e client via connessione TCP, vengono usati dei ByteBuffer locali (sia lato client che lato server) per leggere o scrivere blocchi di byte sul/dal canale. La lunghezza di questi buffer è settata a 4096 per la lettura, a 3072 per la scrittura lato client e a 4096 per la scrittura lato server. Questi numeri sono stati scelti poiché sono abbastanza grandi per permettere ai buffer di contenere tutti i caratteri della richiesta o della risposta (laddove il software venga usato in modo ragionevole). La lunghezza dei buffer di lettura è settata allo stesso numero per entrambe le due entità per evitare che una delle due mandi messaggi che l'altra non è in grado di leggere, completamente, in un solo tentativo. Nello specifico il client è in grado di ricevere risposte di lunghezza massima uguale a quella delle risposte che è in grado di inviare il server. Le sue risposte, però, potrebbero contenere più testo delle richieste che riceve (solo in alcuni casi molto particolari), quindi il client viene forzato ad usare meno testo di quanto sarebbe possibile, per poi assicurare che sia in grado di leggere la risposta del server con una sola chiamata di *read*. Il client, per garantire affidabilità, effettua una gestione del caso limite in cui l'utente prova ad inviare richieste il cui testo superi i 3072 caratteri. Nella fase di registrazione, l'operazione viene impedita e viene stampato un messaggio di errore; negli altri casi la richiesta viene troncata e il server riceverà solo i primi 3072 caratteri.

Gestione eccezioni

Le eccezioni generate dal client e dal server vengono, generalmente, lasciate gestire dalle classi MainClient e MainServer, le quali, richiamando il metodo *start()*, mettono in esecuzione l'uno o l'altro. Tutte le possibili eccezioni vengono catturate all'interno dei due metodi *main()* e gestite in base alle esigenze. Le uniche eccezioni che vengono gestite esattamente dove vengono lanciate, sono quelle che determinano la risposta da inviare al client a, esse infatti vengono gestite in modo tale che si possa offrire un motivo della non riuscita dell'operazione.

Compilazione ed esecuzione

Linux e MacOS

Per compilare ed eseguire i programmi del progetto sono stati creati due file bash (*start_server.sh* e *start_client.sh*) che automatizzano la compilazione e l'esecuzione del server e di un client. Prima di poter eseguire gli script bash potrebbe essere

necessario garantire i permessi di esecuzione ai due file, è possibile farlo con i comandi "*chmod a+x start_server.sh*" e "*chmod a+x start_client.sh*".

Windows

Per la compilazione del server in Windows usare il comando "*javac -cp ../lib/com.fasterxml.jackson/jar_files/jackson-core-2.12.0.jar;../lib/com.fasterxml.jackson/jar_files/jackson-databind-2.12.0.jar;../lib/com.fasterxml.jackson/jar_files/jackson-annotations-2.12.0.jar Card.java MainServer.java NotifyEventInterface.java Progetto.java ServerInterface.java Server.java Utente.java*".

Per l'esecuzione del server usare il comando "*java -cp .;../lib/com.fasterxml.jackson/jar_files/jackson-core-2.12.0.jar;../lib/com.fasterxml.jackson/jar_files/jackson-databind-2.12.0.jar;../lib/com.fasterxml.jackson/jar_files/jackson-annotations-2.12.0.jar MainServer*".

Per la compilazione del client usare il comando "*javac -cp ../lib/com.fasterxml.jackson/jar_files/jackson-core-2.12.0.jar;../lib/com.fasterxml.jackson/jar_files/jackson-databind-2.12.0.jar;../lib/com.fasterxml.jackson/jar_files/jackson-annotations-2.12.0.jar Card.java Client.java MainClient.java NotifyEventInterface.java Progetto.java MessageReader.java ServerInterface.java Utente.java*".

Per l'esecuzione del client usare il comando "*java MainClient*".

Sintassi e spiegazione dei comandi

help: fornisce la sintassi di tutti i comandi disponibili.

register [username] [password]: crea un nuovo utente.

login [username] [password]: esegue il login per l'utente corrispondente all'username digitato.

logout: esegue il logout per l'utente precedentemente connesso.

users: restituisce la lista di tutti gli utenti registrati e del loro stato (online/offline).

online_users: restituisce la lista di tutti gli utenti attualmente online.

list_projects: restituisce la lista di tutti i progetti di cui l'utente è membro.

create_project [project_name]: crea un nuovo progetto e setta l'utente come unico membro.

add_member [project_name] [new_member_username]: aggiunge l'utente digitato tra i membri del progetto.

show_members [project_name]: restituisce la lista di tutti i membri che fanno parte del progetto.

show_cards [project_name]: restituisce la lista di tutte le card del progetto, divise per il loro stato

show_card [project_name] [card_name]: mostra la card selezionata e le sue proprietà.

add_card [project_name] [card_name] [card_description]: aggiunge una card al progetto selezionato.

move_card [project_name] [card_name] [src_state] [dest_state]: sposta la card selezionata dallo stato src_state a quello dest_state.

card_history [project_name] [card_name]: restituisce la lista di tutti gli stati, in ordine temporale, in cui si è ritrovata la card.
read_chat [project_name]: restituisce tutti i messaggi non letti relativi alla chat di progetto selezionata.
send_msg [project_name]: manda un messaggio alla chat di progetto selezionata. Dopo aver eseguito questo comando comparirà la scritta “write your message:”, scrivere di seguito ad essa il messaggio che si vuole inviare.
cancel_project [project_name]: chiude e cancella il progetto selezionato.
quit: per l’uscita dal programma.

Il codice è stato scritto e testato usando la versione di java 12, ma funziona correttamente anche per la versione java 8. Per la serializzazione degli oggetti con json è stata utilizzata la libreria aggiuntiva “Jackson 2.12” (https://github.com/FasterXML/jackson).