Laboratorio di Programmazione di Rete
A.A. 2020/21

Luca Lombardo - Mat. 546688

Indice

Introduzione

1.1 Il progetto

Il programma realizzato consiste in uno strumento con interfaccia a linea di comando per la gestione di progetti collaborativi, ispirato ad alcuni principi della metodologia Kanban. Esso è costituito da dodici file .java, di cui dieci classi e due interfacce:

1. la classe **ServerMain** contiene il metodo main del server. All'interno della classe vengono creati il registro RMI e la ThreadPool di ServerThread;
2. la classe **ServerRemote** estende RemoteObject e contiene i metodi remoti per la registrazione e le callBacks relative ai progetti;
3. la classe **ServerThread** implementa l'interfaccia Runnable e si occupa di ricevere i comandi dal client, eseguirli e restituire i risultati;
4. la classe **User** è una struttura dati contenente nickname e password di un utente;
5. la classe **Project** è una struttura dati contenente le informazioni su un progetto (nome, indirizzo multicast, liste di card);
6. la classe **Card** è una struttura dati contenente le informazioni su una card (nome, descrizione, sequenza di spostamenti tra liste);
7. la classe **ClientMain** contiene il metodo main del client e si occupa di prendere da input i comandi dell'utente, inviarli al server e ricevere i risultati;
8. la classe **ClientRemote** estende RemoteObject e contiene i metodi remoti per la notifica delle callBacks;
9. la classe **ChatThread** implementa l'interfaccia Runnable e si occupa della ricezione di messaggi sulla chat di un progetto;
10. la classe **Message** è una struttura dati contenente le informazioni di un messaggio sulla chat (data, autore e testo).

Il Server

2.1 ServerMain

All'avvio, il server verifica se nella root di esecuzione è presente il file **members.json**: se lo è, lo deserializza utilizzando le librerie Jackson. Questo file contiene le informazioni riguardanti gli utenti registrati al servizio, che vengono memorizzate nell'ArrayList *members* (per la sincronizzazione delle strutture dati non thread-safe si legga più avanti).
Se il file non è presente, esso verrà creato durante l'esecuzione del metodo **register** della classe *ServerRemote*.
Nello stesso path, il server cerca anche la directory **projects**; se la trova, esplora ricorsivamente le sue sotto-directories, aventi come nome il nome di un progetto, e come contenuto i seguenti file json:

- un file *nomeprogettoMembers.json*, contenente le informazioni sugli utenti che sono membri del progetto;
- un file *nomeprogettoMulticast.json*, contenente l'indirizzo ip multicast usato per le comunicazioni sulla chat del progetto;
- un numero variabile di file *nomecard.json*, ognuno dei quali contenente le informazioni su una singola card all'interno del progetto.

Tutti i file elencati vengono deserializzati e le informazioni vengono memorizzate all'interno dell'ArrayList *projects*.
Se, invece, la directory *projects* non è presente, essa viene creata.
Questo meccanismo garantisce la **persistenza** delle informazioni su utenti registrati e progetti attivi, come richiesto dalla specifica del progetto.

Successivamente, viene creato il registro per la **RMI** (porta 6000), sul quale vi è il binding tra il nome "*WORTH*" e lo stub di tipo *ServerRemote*.

Infine, viene creata una **ThreadPool** che esegue un *ServerThread* per ogni connessione con un client. Si è scelto di utilizzare una *FixedThreadPool*: il numero di thread attivi contemporaneamente può essere passato come argomento del programma in maniera da consentire di sfruttare al meglio

l'architettura della macchina. Il valore di default è 8, ideale per un processore con 4 core e HyperThreading.
Il numero di client che tentano di connettersi sarà, con ogni probabilità, molto maggiore del numero di thread, ma la *LinkedBlockingQueue* (lista che può crescere "infinitamente") della FixedThreadPool garantisce che nessun task sarà mai rigettato.

2.2 ServerRemote

La classe *ServerRemote* definisce il tipo dello stub pubblicato sul registro **RMI**.
La classe *ServerMain* passa come argomenti del metodo costruttore le strutture dati *members* e *loggedUsers*, oltre a una variabile usata per la sincronizzazione.

I metodi implementati sono i seguenti:

1. Il metodo remoto **register** permette di registrare un nuovo utente al servizio. Se l'operazione ha successo, sia l'ArrayList *members* che il file json corrispondente vengono aggiornati;

2. Il metodo remoto **registerForCallback** permette al client di registrarsi al meccanismo delle callback per ricevere delle notifiche: tali notifiche riguardano la lista degli utenti registrati e il loro stato (online/offline), ma anche la lista dei progetti dei quali si fa parte;

3. Il metodo remoto **unregisterForCallback** consente al client di rimuovere la propria registrazione alle notifiche (viene invocato in seguito al comando di *logout*);

4. Il metodo **update** (e **doCallbacks**) viene invocato dai *ServerThread* per mandare ai client registrati le notifiche riguardanti gli utenti registrati e il loro stato;

5. Il metodo **addMemberCallback** viene invocato dai *ServerThread* per notificare un client del fatto che è stato aggiunto a un progetto come membro;

6. Il metodo **cancelProjectCallback** viene invocato dai *ServerThread* per notificare i client che erano membri di un progetto del fatto che tale progetto è stato cancellato.

Per mandare le notifiche, la classe necessita delle *ClientRemoteInterface* per ogni connessione. Esse sono memorizzate nella ConcurrentHashMap *clients*.

2.3 ServerThread

La classe *ServerThread* rappresenta il "**task**" eseguito da ogni thread della ThreadPool creata nella classe *ServerMain*.
Gli argomenti passati al costruttore sono:

- l'oggetto *Socket* dato come risultato dalla chiamata del metodo *accept()* sul ServerSocket della classe ServerMain;
- le strutture dati *members*, *loggedUsers* e *projects*;
- il riferimento all'oggetto remoto *service*, utilizzato per invocare i metodi che inviano le notifiche ai client tramite *callback*;
- un array di interi usato come ip multicast "di partenza" per la creazione di nuovi progetti;
- due variabili necessarie per la sincronizzazione.

La classe contiene anche la variabile d'istanza *login* che mantiene le informazioni sulla sessione del client: se tale variabile ha valore *null*, significa che il client non è al momento loggato.

Il metodo **run** implementa un *ciclo*, all'interno del quale si continua a ricevere comandi dal client su una connessione TCP, e sulla stessa connessione si invia il messaggio di successo/errore ed eventualmente i dati da restituire come risultato dell'esecuzione del comando.
Si esce dal ciclo nel caso in cui il client invii il comando *quit* oppure se la connessione dovesse interrompersi. Una volta uscito dal ciclo, il metodo termina e con esso il thread.

2.4 Multi-threading e concorrenza lato Server

Come descritto in precedenza, il Server Worth è un processo **multi-threaded**:

- il **main** è il thread principale, che accetta connessioni TCP sulla porta 10000 e gestisce la ThreadPool;
- un numero n di threads appartengono alla **FixedThreadPool**, dove n è passato come argomento del programma (valore di default: 8), ognuno dei quali gestisce lo scambio di comandi/risposte con un singolo client;
- un certo numero di threads sono attivi per l'**RMI** (la *javadoc* afferma che invocazioni di metodi remoti provenienti da client diversi sono eseguite da thread diversi, mentre in caso di invocazioni concorrenti provenienti dallo stesso client non vi è alcuna garanzia).

Le operazioni su strutture dati **non thread-safe**, ovvero *members*, *loggedUsers* e *projects* (di tipo ArrayList), nonché quelle sui file json, avvengono solo ed esclusivamente all'interno di blocchi sincronizzati esplicitamente su una delle due variabili apposite (*userSync* e *projectSync*), i cui riferimenti vengono passati come parametri alle classi ServerRemote e ServerThread.

L'unica variabile che utilizza una struttura dati concorrente è *clients*, all'interno della classe ServerRemote. Tale variabile è di tipo **ConcurrentHashMap**; le operazioni di aggiornamento vengono eseguite tramite il metodo *putIfAbsent* per garantire l'atomicità delle operazioni composte ed evitare *race conditions*.

Il Client

3.1 ClientMain

La classe *ClientMain* contiene numerose variabili d'istanza, delle quali:

- *serverObject*, *remoteObject*, *callbackObj*, *stub* sono le variabili utilizzate per l'**RMI**;
- *members* e *loggedUsers* sono le liste degli utenti rispettivamente registrati e loggati; esse vengono ricevute la prima volta tramite la connessione TCP in seguito a un login eseguito con successo, e poi vengono aggiornate tramite il meccanismo delle **callback** (classe *ClientRemote*);
- *multicastIp* e *chat* sono mappe utilizzate per gestire le **chat** dei progetti;
- *ds* è il **DatagramSocket** utilizzato per inviare messaggi sulle chat dei progetti. La porta viene scelta cercando la prima disponibile a partire dalla porta 11000.

Dopo aver ottenuto l'accesso al registro RMI e aver instaurato la connessione TCP con il server (porta 10000), il client entra in un **ciclo**: uno scanner prende un comando da input, esso viene inviato al server e si resta in attesa di una risposta.

In seguito a un'operazione di login andata a buon fine, il client crea un **ChatThread** per ogni progetto; tali threads notificheranno il client dei messaggi ricevuti tramite il meccanismo delle *callback*.
Qualora fosse necessario creare nuovi ChatThread (nel caso in cui si venga aggiunti come membro di un progetto) o terminarne alcuni (se un progetto viene cancellato), queste operazioni vengono svolte dalla classe *ClientRemote*.

3.2 ClientRemote

Il costruttore della classe *ClientRemote* riceve dal ClientMain:

- le variabili *members*, *loggedUsers*, *multicastIp* e *chat* descritte in precedenza;
- la mappa *chatThreads*, che contiene le associazioni fra thread e *MulticastSocket*;
- una variabile per la sincronizzazione.

La classe implementa i seguenti metodi remoti:

- **notifyEvent** riceve come parametri le liste di utenti registrati e loggati (ogni volta che esse variano nel server) e ne aggiorna le copie locali del client;
- **notifyProject** è invocato dal server quando il client viene aggiunto a un progetto. Al suo interno viene creato un nuovo ChatThread e la mappa *chatThreads* viene aggiornata di conseguenza;
- **notifyCancelProject**, infine, viene invocato dal server quando un progetto del quale il client era membro viene cancellato. Questo metodo chiude il *MulticastSocket* del ChatThread corrispondente, che si auto-interromperà in seguito a questo evento. La mappa *chatThreads* viene poi aggiornata.

3.3 ChatThread

Il costruttore della classe *ChatThread* prende come parametri l'indirizzo ip di un progetto (espresso come stringa), la mappa che contiene le associazioni tra progetti e chat, e un MulticastSocket.

L'unico metodo implementato è **run**, all'interno del quale si entra in un ciclo dove si esegue una **receive** sul MulticastSocket passato. Ogni qualvolta si riceve un datagramma, la struttura dati contenente la chat viene aggiornata col messaggio ricevuto.

Quando un ChatThread deve essere terminato, il suo MulticastSocket viene chiuso (dalle classi ClientMain o ClientRemote): quando ciò avviene, viene sollevata un'eccezione del tipo *SocketException*, catturata da un try-catch che si occupa anche di interrompere il thread stesso.

3.4 Multi-threading e concorrenza lato Client

Così come il Server, anche il Client Worth è un processo **multi-threaded**:

- il **main** è il thread principale e gestisce la connessione TCP con il Server, nonché lo scambio di comandi e risposte con esso;

- dopo il login, n thread sono sempre attivi per gestire le **chat** dei progetti (n è quindi il numero di progetti dei quali si è membri);
- un thread **RMI** è attivo per il meccanismo delle callback.

Esattamente come nel Server, l'accesso e le operazioni su strutture dati **non thread-safe** (*members* e *loggedUsers*) avvengono esclusivamente all'interno di blocchi sincronizzati esplicitamente sulla variabile *userSync*, condivisa tra ClientMain e ClientRemote.

Le variabili *multicastIp* e *chat*, invece, sono implementate tramite una **ConcurrentHashMap**: esse sono condivise fra tutti i thread attivi (ClientMain, ClientRemote e n ChatThread). L'atomicità delle operazioni composte viene garantita utilizzando il metodo *putIfAbsent* per ogni aggiornamento.

Istruzioni

Nota: il copia-incolla di comandi multi-riga sul terminale può non funzionare correttamente. Si consiglia di copiare i comandi dal file *istruzioni.txt*.

4.1 Compilazione

Il progetto utilizza *jackson-annotations-2.9.7.jar*, *jackson-core-2.9.7.jar* e *jackson-databing-2.9.7.jar* per gestire serializzazione e deserializzazione.
Per compilare Server e Client è necessario usare uno dei seguenti comandi, a seconda della propria piattaforma:

Linux

javac -classpath .:jackson-core-2.9.7.jar:jackson-databind-2.9.7.jar:jackson-annotations-2.9.7.jar Card.java ChatThread.java ClientMain.java ClientRemote.java ClientRemoteInterface.java Message.java Project.java ServerMain.java Server-Remote.java ServerRemoteInterface.java ServerThread.java User.java

O semplicemente:

*javac -classpath .:jackson-core-2.9.7.jar:jackson-databind-2.9.7.jar:jackson-annotations-2.9.7.jar *.java*

Windows

javac -classpath .;jackson-core-2.9.7.jar;jackson-databind-2.9.7.jar;jackson-annotations-2.9.7.jar Card.java ChatThread.java ClientMain.java ClientRemote.java ClientRemoteInterface.java Message.java Project.java ServerMain.java Server-Remote.java ServerRemoteInterface.java ServerThread.java User.java

O semplicemente:

*javac -classpath .;jackson-core-2.9.7.jar;jackson-databind-2.9.7.jar;jackson-annotations-2.9.7.jar *.java*

4.2 Esecuzione

I comandi necessari per l'esecuzione variano a seconda della propria piattaforma.

Linux

Per eseguire il Server Worth:

java -cp .:jackson-core-2.9.7.jar:jackson-databind-2.9.7.jar:jackson-annotations-2.9.7.jar ServerMain

È possibile passare come argomento del server un intero, che verrà utilizzato come numero di threads della FixedThreadPool. Se non viene passato alcun argomento, verrà utilizzato il numero di default (8).

Per eseguire il Client Worth:

java -cp .:jackson-core-2.9.7.jar:jackson-databind-2.9.7.jar:jackson-annotations-2.9.7.jar ClientMain

Windows

Per eseguire il Server Worth:

java -cp .;jackson-core-2.9.7.jar;jackson-databind-2.9.7.jar;jackson-annotations-2.9.7.jar ServerMain

È possibile passare come argomento del server un intero, che verrà utilizzato come numero di threads della FixedThreadPool. Se non viene passato alcun argomento, verrà utilizzato il numero di default (8).

Per eseguire il Client Worth:

java -cp .;jackson-core-2.9.7.jar;jackson-databind-2.9.7.jar;jackson-annotations-2.9.7.jar ClientMain

4.3 Comandi

Di seguito sono elencati tutti i comandi disponibili sul Client Worth. I comandi sono **case-sensitive**. È possibile visualizzare una lista simile tramite il comando *help* direttamente all'interno del programma.

- **help**
  Stampa a schermo la schermata di aiuto, contenente sintassi e semantica dei comandi disponibli.

- **register** *nickUtente password*
  Registra un nuovo utente al servizio.
- **login** *nickUtente password*
  Effettua il login per accedere al servizio.
- **logout** *nickUtente*
  Effettua il logout dell'utente dal servizio.
- **listUsers**
  Stampa a schermo la lista degli utenti registrati al servizio e il loro stato (online/offline).
- **listOnlineUsers**
  Stampa a schermo la lista degli utenti online in questo momento.
- **listProjects**
  Stampa a schermo la lista dei progetti dei quali si è membri.
- **createProject** *projectName*
  Crea un nuovo progetto.
- **addMember** *projectName nickUtente*
  Aggiunge un utente a un progetto.
- **showMembers** *projectName*
  Stampa a schermo la lista dei membri di un progetto.
- **showCards** *projectName*
  Stampa a schermo la lista di card associate ad un progetto.
- **showCard** *projectName cardName*
  Stampa a schermo le informazioni (nome, descrizione, lista) di una card associata a un progetto.
- **moveCard** *projectName cardName listaPartenza listaDestinazione*
  Sposta una card di un progetto da una lista a un'altra.
- **getCardHistory** *projectName cardName*
  Stampa a schermo la storia della card, ovvero la sequenza di spostamenti da una lista a un'altra.
- **readChat** *projectName*
  Stampa a schermo i messaggi della chat di un progetto.
- **sendChatMsg** *projectName messaggio*
  Invia un messaggio alla chat di un progetto.
- **cancelProject** *projectName*
  Cancella un progetto.

Luca Lombardo
Mat. 546688
29/11/2020