Relazione Progetto WORTH

Teodorico Pacini, 578987

Università di Pisa

Premesse

L'implementazione dell'ambiente client-server che vedrete in questo progetto sarà basata su alcuni punti chiave:

- I client sono pensati per essere eseguiti da macchine diverse, così come il server che avrà una propria macchina dedicata.
- Il server resterà sempre attivo e potrà essere disattivato a causa di un'anomalia o di un errore imprevisto (per testare il funzionamento della persistenza si può utilizzare qualsiasi segnale che faccia terminare il thread *(SIGINT, SIGKILL ecc..)).*
- Gli utenti potranno utilizzare solo il client da me creato.
- La cartella su cui avverrà la persistenza dei dati deve chiamarsi "*SavedState*" e trovarsi nella directory principale del progetto. *(volendo il nome e la posizione della directory si possono modificare andando a gestire la stringa MAIN_PATH che si trova nella classe DBMS.java)*

P.S. Sia i client che il server sono stati testati utilizzando la stessa macchina ("localhost").

Architettura generale

Il sistema è strutturato in due macro componenti, la parte relativa al *client (ClientMain.java)*, la quale viene utilizzata dall'utente per poter interagire con il sistema, e la parte *server (ServerMain.java)* che gestisce le richieste dei client e restituisce una risposta riguardo l'esito dell'operazione effettuata.

Client e server comunicano attraverso una connessione TCP che utilizza un'interfaccia basata su *SocketChannel*.

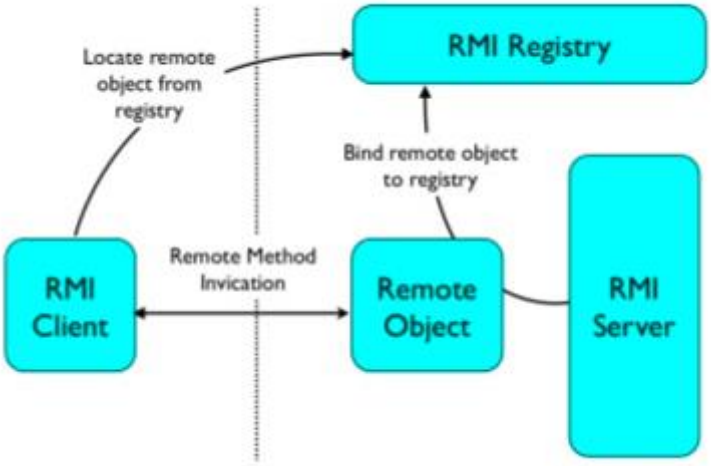
Il **server** implementa il *multiplexing dei canali mediante NIO* utilizzando un selettore che garantisce la gestione delle richieste dei client in maniera veloce e senza generare carico eccessivo su CPU e RAM.

In questo progetto ho preferito adottare un'*architettura NIO multiplexed* piuttosto che un approccio *multithreaded*, poiché il primo consente la gestione di più connessioni utilizzando un solo thread, mentre nel secondo caso è necessario l'avvio di numerosi thread con una conseguente maggior richiesta di potenza di calcolo e un aumento della memoria occupata dal "programma"; inoltre, questo ultimo fattore peggiora ancor di più nel caso in cui si voglia scalare l'intero sistema.

Il primo vantaggio di questa scelta, quindi è un minor consumo di CPU e RAM, e inoltre una maggiore capacità di scalare.

Tutto ciò giustamente garantisce un tempo di risposta molto rapido, che viene mantenuto all'aumentare dei client. Nel caso in cui un unico thread non bastasse per gestire tutte le richieste, si possono comunque utilizzare più thread mantenendo la stessa efficienza.

Per quanto riguarda la registrazione, il **client** e il server utilizzano un'interfaccia condivisa (*RMIRegistrationInterface.java*), mentre per il meccanismo di callback si sfruttano due interfacce (una del server (*RMICallbackInterface.java*) e una del/i client (*ClientNotifyInterface.java*)).



Il Remote Object rappresenta RMICallbackImpl.java

(e il corrispettivo per la registrazione)

Le restanti operazioni (tranne *listUsers*, *listOnlineUsers* e la scrittura/lettura sulla *chat*) vengono eseguite sulla connessione TCP che viene instaurata all'avvio del client (*N.B. viene creata all'avvio ma il processo di registrazione non la utilizza*).

Per svolgere queste operazioni l'utente può usare i comandi offerti dalla **CLI** (che si possono visualizzare tramite il comando "*help*"), scrivendoli da linea di comando una volta comparso il simbolo *">"*. Il client stamperà a schermo le risposte del server che possono consistere in *messaggi d'errore* o *messaggi di successo* (es. *200 OK*), seguiti dall'esito prodotto dal comando richiesto.

```
> show_cards p1
Nome card                       |Stato
spesa                           |todo
comprare_il_latte               |todo
```

Esempio di richiesta/risposta lato client

Gli unici vincoli che ho posto all'input sono:

- Il nome del progetto non deve contenere spazi e nel caso in cui venga inserita una stringa contenente spazi, quest'ultima verrà interpretata come due stringhe distinte.
- Stessa cosa per il nome delle card.
- Per il resto, possono contenere spazi: la *descrizione della card* e i *messaggi inviati nella chat*.

La modalità con cui ho scelto di **comunicare indirizzo e porta di multicast** al client è tramite la richiesta esplicita ("*get_parameter*") come con gli altri comandi ma, a differenza di questi ultimi, non si trova tra le opzioni disponibili nella CLI ma è un comando che viene utilizzato dal metodo *checkNotifications.*

Riguardo la **persistenza** ho scelto di scrivere i dati in file json, tramite l'utilizzo di una libreria esterna *json-simple* (ver. 1.1). Questa libreria offre un meccanismo intuitivo e veloce per scrivere e recuperare dati da un file json per questo l'ho preferita rispetto a GSON o Jackson, anche se quest'ultime risolvono il problema presentato sotto.

Non ho scelto di serializzare i dati tramite l'interfaccia *Serializable* poiché ho ritenuto quest'ultimo essere un approccio troppo dispendioso riguardo il consumo di risorse.

P.S. Le diciture "@SuppressWarnings("unchecked")*" che si possono trovare nei metodi che utilizzano questa libreria esterna, sono dovuti all'utilizzo da parte di quest'ultima di "*parametri non generici*".*

Da quel che ho potuto testare, non ho riscontrato comunque errori durante l'utilizzo.

Thread, strutture dati e concorrenza

I thread che vengono attivati sono *uno* per il server (*ServerMain.java*), e *uno* per ogni client (*ClientMain.java*).

Il **server** utilizza tre strutture dati principali all'interno della sua classe:

- ***ArrayList<Project> projects***: rappresenta la lista dei progetti creati dai vari utenti.
- ***HashMap<String, String> users***: contiene la lista degli utenti registrati a cui è associato lo stato in cui si trovano (online o offline).
- ***HashMap<String, String> credentials***: contiene la lista degli utenti registrati a cui è associata la password.

Tutte queste strutture dati non implementano nessun meccanismo di concorrenza, così come i metodi all'interno della classe *ServerMain.java* che non utilizzano la keyword *synchronized*, e questo poiché l'utilizzo di un Selector non conduce solitamente a situazioni di concorrenza sui dati.

Lato server l'unico controllo che ho ritenuto necessario è nella classe *DBMS.java.* In questo caso il metodo *existUser.java* e *registerUser.java* possono venir invocati da più thread e quindi portare a un'inconsistenza dei dati, per questa ragione ho utilizzato *synchronized* che gestisce la concorrenza nell'utilizzo del metodo.

Per quanto riguarda il lato client, non ho implementato nessun controllo di concorrenza, poiché ogni client ha la propria l'interfaccia remota (*ClientNotifyImpl.java*) mentre la classe principale (*ClientMain.java*) non richiede nessun meccanismo di gestione della concorrenza.

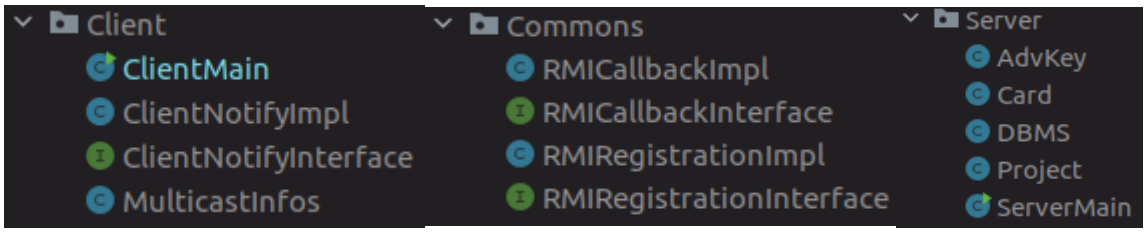
A differenza del server, il **client** utilizza solamente due strutture dati principali:

- ***HashMap<String, MulticastInfos> projects***: contiene la lista dei progetti di cui l'utente fa parte, associati alle informazioni di multicast che sono utilizzate per poter partecipare al "*gruppo di multicast*".
- ***HashMap<String, String> local_users***: contiene le associazioni nome utente – stato, aggiornate dal meccanismo di callback e visualizzate con *listUsers* e *listOnlineUsers.*

Infine, delle due interfacce RMI, di registrazione e di callback, solo la seconda utilizza *synchronized* per gestire la concorrenza sui metodi, *registerForCallback* e *unregisterForCallback,* utilizzati dai client, e su *update,* utilizzato da DMBS e Server.

Descrizione delle classi

La directory principale è divisa in tre sotto directory: *src/Client*, *src/Server* e *src/Commons*.



Client

Questa directory contiene tutte le classi utilizzate dal Client (*senza contare le classi che gestiscono il meccanismo di RMI, che sono condivise da Client e Server*):

- **ClientMain.java**: Questa è la classe principale del client, contiene i vari metodi per gestire le richieste dell'utente (*login, listProjects, addMember...)* e inoltre implementa dei metodi ausiliari come il *parser* delle richieste, il controllore di nuove aggiunte o rimozioni da progetti (*checkNotifications*), o il metodo che permette di visualizzare la lista degli utenti, salvata localmente in una struttura dati all'interno di

ClientNotifyImpl.java. Il lavoro di cui si occupa la classe è quello di creare una connessione con il server e successivamente gestire le richieste dell'utente tramite l'analisi dell'input e l'invio di richieste al server.

- **ClientNotifyImpl.java**: Questa è la classe relativa all'oggetto RMI utilizzato per gestire le notifiche del server (*callback*). Oltre ad avere un *HashMap* utilizzata per mantenere una copia locale della lista degli utenti, presenta anche l'implementazione dei metodi invocati dal server.
- **ClientNotifyInterface.java**: Questa è l'interfaccia dell'oggetto RMI che sarà "*registrato nel registry*" e utilizzato dal client per ricevere la lista degli utenti registrati aggiornata.
- **MulticastInfos.java**: Questa è la classe utilizzata per memorizzare porta, indirizzo e socket multicast utilizzata dal client per comunicare con la chat del progetto. Presenta solo tre metodi *getters* che restituiscono rispettivamente *MulticastSocket, indirizzo* e *porta*.

Server

Questa directory contiene tutte le classi utilizzate dal Server:

- **ServerMain.java**: Questa è la classe principale del server, contiene i metodi per effettuare *login* e *logout*, i vari metodi che eseguono le richieste dell'utente (*moveCard, cancelProject, showCard....),* i metodi per il ripristino e la persistenza di progetti e card sul disco e infine dei metodi ausiliari che possono svolgere la configurazione della *comunicazione TCP*, l'*inizializzazione e esportazione degli oggetti RMI* o la generazione di un nuovo *indirizzo di multicast.*
  All'avvio il server ripristina i dati che si trovano sul disco, crea e configura i due servizi RMI dopodiché crea la SocketChannel su cui si collegheranno i client e crea un selettore che si occuperà di gestire i vari channel.
- **AdvKey.java**: Questa classe contiene tre stringhe, *nickname*, *request* e *response.* Questo permette al server di poter creare un'istanza di questa classe e poter associarla ad una *key* tramite il metodo *attach.*
- **Card.java**: Questa classe rappresenta le card che possono far parte di un progetto. Presenta tre stringhe (*name*, *description* e *list)* che individuano rispettivamente il nome, la descrizione della card e la lista in cui si trova attualmente. Inoltre *movements* è un'ArrayList che contiene al suo interno i nomi delle liste visitate dalla card (ordinate). Oltre ad avere dei metodi *getters* e *setters*, la classe possiede un altro metodo (*changeList)* che gestisce lo spostamento di una card da una lista ad un'altra.
- **Project.java**: Questa classe contiene i metodi e le strutture dati necessarie per la gestione dei progetti. Presenta i metodi necessari ad aggiungere nuovi membri o nuove card, visualizzare tutte le card o tutti i membri, visualizzare una specifica card

e spostarla da una lista ad un'altra. Inoltre, presenta un metodo *isDone* che dice al server se tutte le card si trovano nella lista done (*e quindi può procedere alla cancellazione del progetto*), e vari metodi *setters* e *getters.*

- **DBMS.java**: Questa classe si occupa di registrare nuovi utenti al "servizio" e di garantire la persistenza delle registrazioni nel disco. Per svolgere queste mansioni utilizza i metodi *existUser* e *registerUser* per la registrazione, mentre *restoreRegistration* e *updateRegistration* per la persistenza sul disco. Infine la classe implementa anche un metodo *setLocal_ref* che permette di ottenere il riferimento all'RMI callback, cosicché ad ogni registrazione il DBMS possa aggiungere il nuovo utente alla lista (degli utenti registrati).

Commons

Questa directory contiene tutte le classi condivise da Client e Server:

- **RMICallbackImpl.java**: Questa classe gestisce il meccanismo di RMI Callback, presenta due metodi invocati dai client per registrarsi o annullare la registrazione alle callback. Possiede due strutture dati, una, *clients,* per tener traccia dei client che si registrano alle callback, su cui poi effettuare le stesse; la seconda, *users,* rappresenta la lista degli utenti registrati da inviare ai client. I due metodi restanti (*update*, *doCallbacks*) sono utilizzati, per aggiornare la lista degli utenti o aggiungere un solo utente, in caso di *login, logout o register.*
- **RMICallbackInterface.java**: Questa è l'interfaccia dell'oggetto RMI utilizzato per le callback. Possiede due campi *PORT* e REMOTE_OBJECT_NAME, utilizzati per creare il bind del remote object con un registry e per fare la locate dello stesso.
- **RMIRegistrationImpl.java**: Questa classe gestisce il meccanismo di RMI per le registrazioni degli utenti. Implementa il metodo *register* che controlla la validità di username e password dopodiché esegue dei controlli invocando i metodi del DBMS, e in base all'esito restituisce la risposta all'utente.
- **RMIRegistrationInterface.java**: Questa è l'interfaccia dell'oggetto RMI utilizzato per le registrazioni degli utenti. Possiede due campi *PORT* e REMOTE_OBJECT_NAME, utilizzati per creare il bind del remote object con un registry e per fare la locate dello stesso.

Istruzioni

Il progetto utilizza una sola libreria esterna, *json-simple,* che si trova già all'interno della directory del progetto.

Per la compilazione ho creato due script bash:

- ***clean.sh***: elimina ogni file ausiliario creato durante le vecchie compilazioni del progetto. *Non è necessario eseguirlo prima di compilare, poiché verrà utilizzato da compile.sh*
- ***compile.sh*****:** esegue *clean.sh* e compila l'intero progetto. Nel caso vengano generati degli errori durante la sua esecuzione, lo script restituisce dei messaggi evidenziati dal tag *[ERROR]*.

Compilazione ed esecuzione

*N.B. Prima di eseguire qualsiasi processo di compilazione/esecuzione, bisogna posizionarsi all'interno della directory "**src**"*.

Per compilare l'intero progetto, eseguire i seguenti comandi da terminale (*dalla directory corretta indicata nella riga sopra*):

1. chmod +x compile.sh
2. ./compile.sh

Se non vengono visualizzati messaggi di errore allora la compilazione è avvenuta con successo.

Per poter eseguire i client ed il server:

- ***Client***: java Client.ClientMain localhost 4382
- ***Server***: java -cp :../json-simple-1.1.jar Server.ServerMain 4382

Nel caso dei client, *localhost* indica l'indirizzo IP del server mentre *4382* indica la porta stabilita per poter connettersi al server.

Nel caso del server, *4382* indica la porta designata per la comunicazione con i vari client.