Multi-User Chat

Course	Distributed Programming
Date	@October 3, 2023
Type	Exam
Status	Not Started

Descrizione

L’applicazione ***Multi-User Chat*** presenta una struttura di tipo *client-server*, dove il ruolo dei client viene svolto dagli utenti dell’applicazione.

Requisiti del server

Il server *deve*:

[FS1] poter accettare nuove richieste di connessione che arrivano dagli utenti,

[FS2] poter gestire le disconnessioni degli utenti,

[FS3] poter inviare in broadcast i messaggi inviati da un utente,

[FS4] poter inviare ad un altro utente i messaggi inviati da un utente,

[FS5] mostrare nella propria console i messaggi inviati dai client,

[FS6] gestire disconnessioni brusche dei client.

Requisiti del client

Il client *deve*:

[FC1] potersi connettere al server usando il proprio indirizzo IP e numero di porta,

[FC2] poter inserire un username,

[FC3] poter inviare messaggi in broadcast,

[FC4] poter inviare messaggi ad un altro utente,

[FC5] visualizzare i messaggi ricevuti dagli altri utenti,

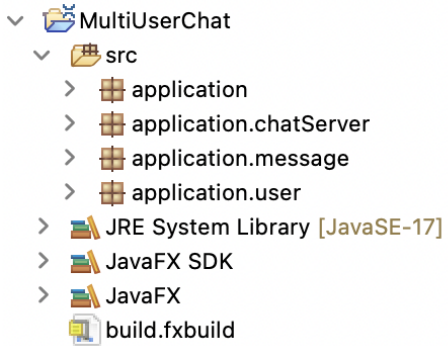
[FC6] potersi disconnettere,

[FC7] gestire disconnessioni brusche del server.

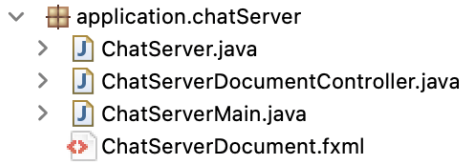
Struttura

Si è deciso di fornire un'interfaccia grafica basilare sia per il client, che per il server. A tale scopo, per entrambe le parti si è adottata la struttura MVC (Model View Controller), usando SceneBuilder per ottenere il file Document in formato FXML che descrive l'interfaccia, mentre il Controller è stato ottenuto per generazione automatica dell'IDE Eclipse stesso a partire dal Document. Per rappresentare i messaggi scambiati tra i client e il server e viceversa, è stata predisposta l'apposita classe `Message`.

In generale abbiamo la cartella `chatServer` che contiene tutto ciò che riguarda il server, la cartella `message`, che contiene ciò che riguarda i messaggi scambiati e la cartella `user`, che invece contiene ciò che riguarda i client.



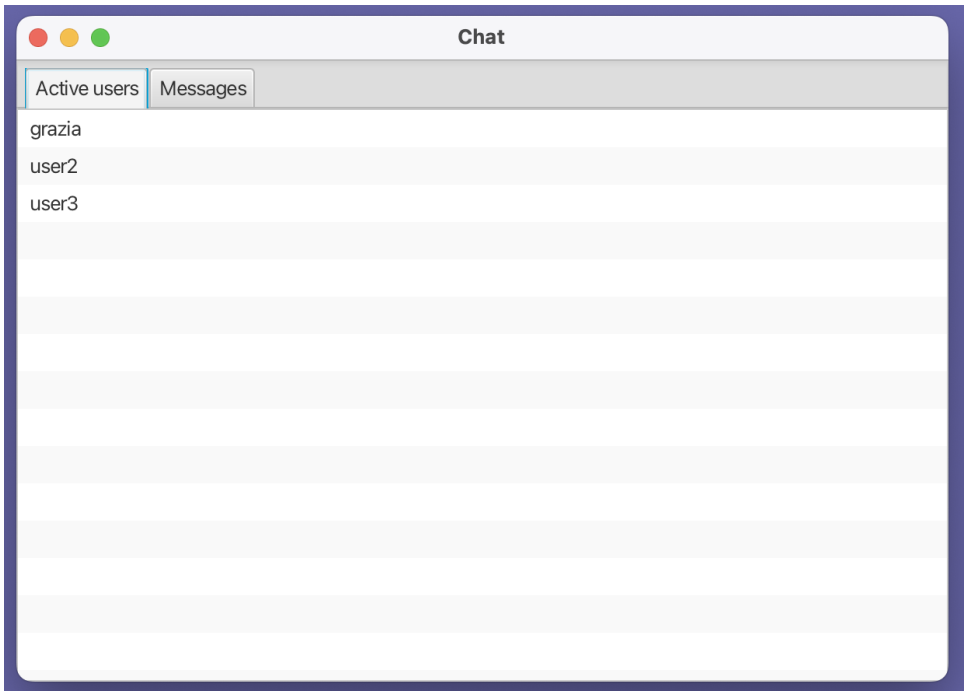
Server



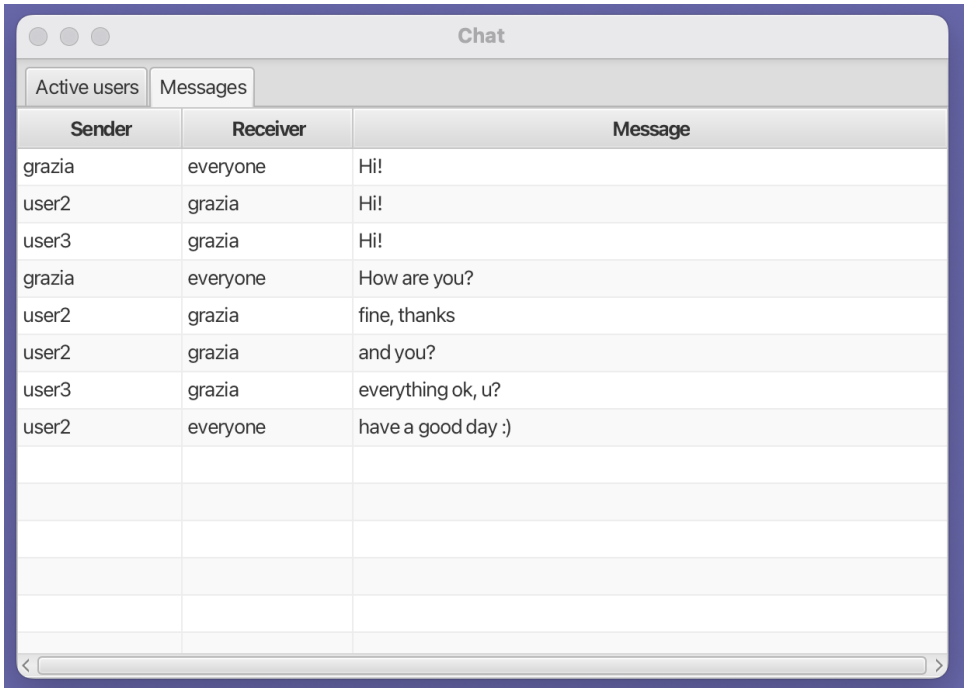
Main

La classe `ChatServerMain` consente di eseguire il lato server dell'applicazione. Avviando il server, notiamo la presenza di un pannello con due tab:

- *active users*, che mostra gli utenti attivi in tempo reale,



- *messages*, che mostra i messaggi che sono stati inviati dagli utenti al server.



I messaggi che arrivano al server, sono anche stampati all'interno della console di sistema.

Logica

La classe `ChatServer`, contiene la logica del server e consente di:

- creare un'istanza del server usando il costruttore `ChatServer( int port)`,
- accettare nuove connessioni dei client tramite il metodo `acceptConnection()`, il server è multi-threaded e, per ogni connessione del client, viene creata una nuova istanza del server usando il costruttore `ChatServer( String username, ObjectInputStream is)`,
- disconnettere un utente tramite il metodo `disconnectUser()`,
- inviare un messaggio in broadcast tramite il metodo `broadcast()`,
- inviare un messaggio a un utente tramite il metodo `oneToOne()`,
- leggere i messaggi in arrivo tramite il metodo `readIncomingMessages()`.

Ogni thread creato per gestire le richieste di uno specifico client, nella propria implementazione del metodo `run()` si occupa di:

- effettuare la lettura dei messaggi in arrivo ed inoltrarli ai destinatari (a seconda che sia *one to one*, oppure in *broadcast*),
- se durante la lettura si verifica una `EOFException` allora significa che il client ha richiesto/subito una disconnessione, dunque viene disconnesso tramite l'apposito metodo.

Strutture e attributi

Abbiamo come attributi *statici* anzitutto le strutture che devono essere condivise da tutte le istanze del server che sono in esecuzione, in particolare abbiamo:

- la lista osservabile `console` che contiene i messaggi ricevuti dal server,
- la lista osservabile `activeUsers` che contiene gli username degli utenti attivi,
- una hash map `users` che contiene la corrispondenza tra l'username di un utente attivo e la socket associata,
- una hash map `connections` che contiene la corrispondenza tra una socket e l'output stream associato (questa si è resa necessaria perché si verificava un'eccezione di tipo *stream corrupted*, dovuta al tentativo di aprire due stream verso lo stesso client e che quindi avevano lo stesso header AC, mentre mantenendo la corrispondenza nella mappa, se lo stream verso quel client è già stato aperto, è sufficiente andarlo a recuperare).

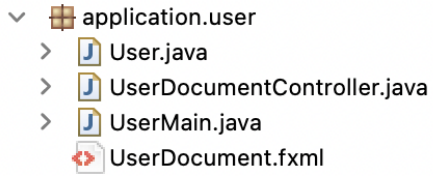
Oltre le strutture, è comune a tutte le istanze della classe la `ServerSocket`.

Dopodiché, ogni thread creato nel metodo `acceptConnection()`, ha il proprio input stream e l'username dell'utente con cui sta comunicando.

Controller

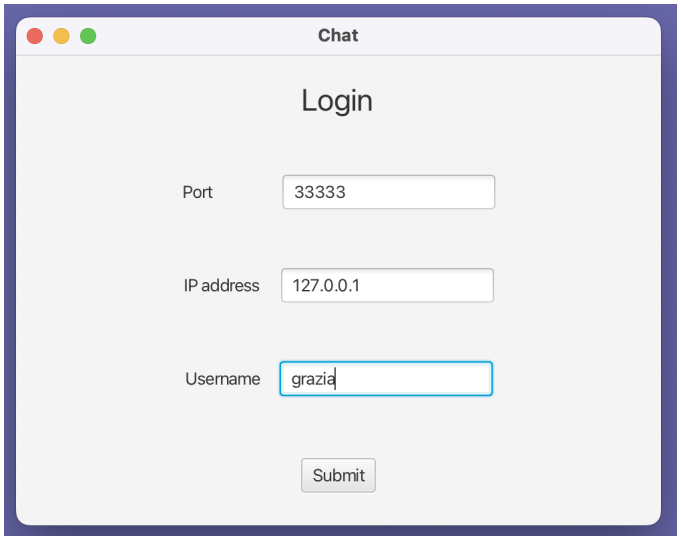
La classe `ChatServerDocumentController`, contiene il metodo per l'inizializzazione dell'interfaccia. All'interno del metodo viene creata l'istanza del `ChatServer`, vengono associate le liste osservabili contenenti i messaggi e gli utenti attivi rispettivamente alla TableView ed alla ListView (in questo modo le modifiche apportate sulle strutture si propagano in tempo reale nella GUI) e viene avviato un Task che viene eseguito in parallelo al thread principale che si occupa invece della gestione dell'interfaccia. Il Task non fa altro che eseguire in loop il metodo `acceptConnection()` del server.

Client



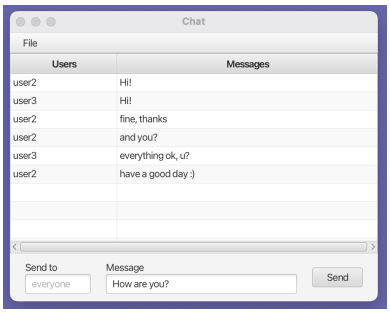
Main

La classe `UserMain` consente di eseguire il lato client dell'applicazione. Avviando il client, abbiamo un primo pannello di login che consente di inserire il numero di porta, l'indirizzo IP (entrambi che servono per creare la Socket del client) e l'username.

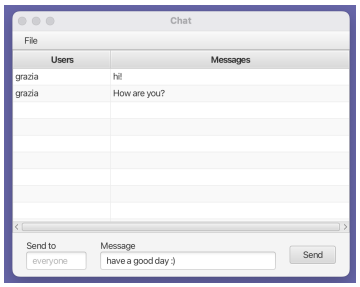


Se i campi vengono lasciati vuoti, o non sono conformi a quanto atteso, vengono mostrati dei messaggi di errore. Se la connessione non va a buon fine, viene mostrato un messaggio di errore e l'utente può riprovare ad effettuare la connessione.

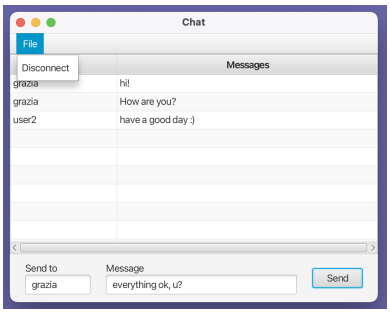
Una volta effettuata la connessione, appare un secondo pannello che mostra i messaggi ricevuti dall'utente, consente all'utente di inviare messaggi a un altro utente o a tutti gli utenti attivi e di disconnettersi tramite l'apposito item della menu bar posta in alto.



User *grazia*.



User *user2*.



User *user3*.

Logica

La classe `User` contiene la logica del client e consente di:

- creare un'istanza del client usando il costruttore `User(String IPAddress, int portNumber)`,
- connettersi al server usando il metodo `connect()`,
- disconnettersi dal server usando il metodo `disconnect()`,
- inviare un messaggio in broadcast usando il metodo `sendMessage(String message)`,
- inviare un messaggio ad un altro utente usando il metodo `sendMessage(String receiver, String message)`,
- leggere i messaggi in ingresso usando il metodo `readIncomingMessages()`.

Quando il client si collega al server, viene creato un nuovo thread, il cui compito è quello di leggere i messaggi in ingresso e mostrarli a video fino alla disconnessione dell'utente.

Strutture e attributi

All'interno di della classe manteniamo una lista osservabile di messaggi, che verranno poi mostrati all'interno della TableView, che è nota anche al thread che esegue le letture (essendo

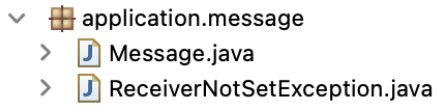
statica). L’username, la socket e gli stream di input e di output sono comuni a tutte le istanze, mentre indirizzo IP, numero di porta e il thread che viene avviato, dipendono dall’istanza.

Controller

La classe `UserDocumentController` contiene il metodo per inizializzare l’interfaccia, che si occupa di impostare come primo pannello visibile quello di login e di impostare le colonne della TableView per la visualizzazione degli attributi di interesse.

Dopodiché abbiamo un handler associato al Submit button del login che consente di provare ad effettuare la connessione al server, un handler associato al Disconnect item della Menu Bar ed un handler associato al Send button del pannello per la visualizzazione dei messaggi.

Message



È la classe usata per descrivere il messaggio. Un messaggio ha un `sender`, un `receiver` ed un `message`, che rappresenta il testo del messaggio inviato.

Vengono messi a disposizione due costruttori:

- `Message(String sender, String message)` per i messaggi da inviare in broadcast, per cui non c’è un receiver particolare,
- `Message(String sender, String receiver, String message)` per i messaggi da inviare ad un receiver in particolare.