Continguts

1 Introducció **1**
1.1 Funcionament de l'aplicació 2
1.2 Estructura de l'aplicació 3
1.2.1 Estructura del mòdul servidor 3
1.2.2 Estructura del mòdul client 4

1 Introducció

En aquesta practica anem a implementar una aplicació de missatgería amb una arquitectura client-servidor, on la comunicació es realitzarà mitjançant sockets i on el protocol de comunicació es basarà en missatges en format JSON.

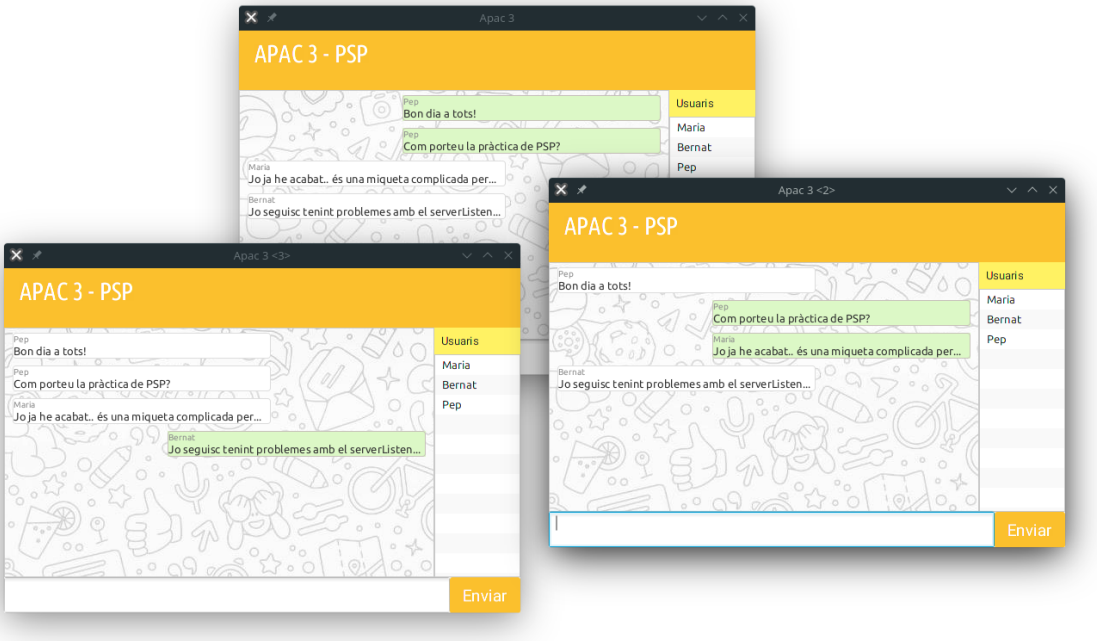


Figura 1: Exemple d'aplicació

Per a això se us proporcionarà ja el projecte de base, amb la interfície del client i la seua funcionalitat ja implementada, de manera que només heu de completar la part de comunicacions. De tota manera, el codi està documentat, per si voleu donar-li una ullada per vore com està fet.

El projecte és un projecte Gradle que consta de dos mòduls, la part client i la part servidor. Per a això, a l'arrel disposeu del fitxer *settings.gradle* amb el següent contingut:

```
rootProject.name = 'PSPMessages'
include('client', 'server')
```

On s'especifica que el projecte consta de dos mòduls a les carpetes *client* i *server*.

Per executar cadascun dels mòduls es farà indicant el mòdul abans del run, de la següent manera:

- Per arrancar el servidor:

```
gradle server:run
```

- Per arrancar el client, al que caldrà proporcionar-li l'adreça IP del servidor i el nom d'usuari:

```
gradle client:run --args "127.0.0.1 Pep"
```

En cas que l'usuari ja estiga registrat al servidor, no deixarà obrir l'aplicació.

1.1 Funcionament de l'aplicació

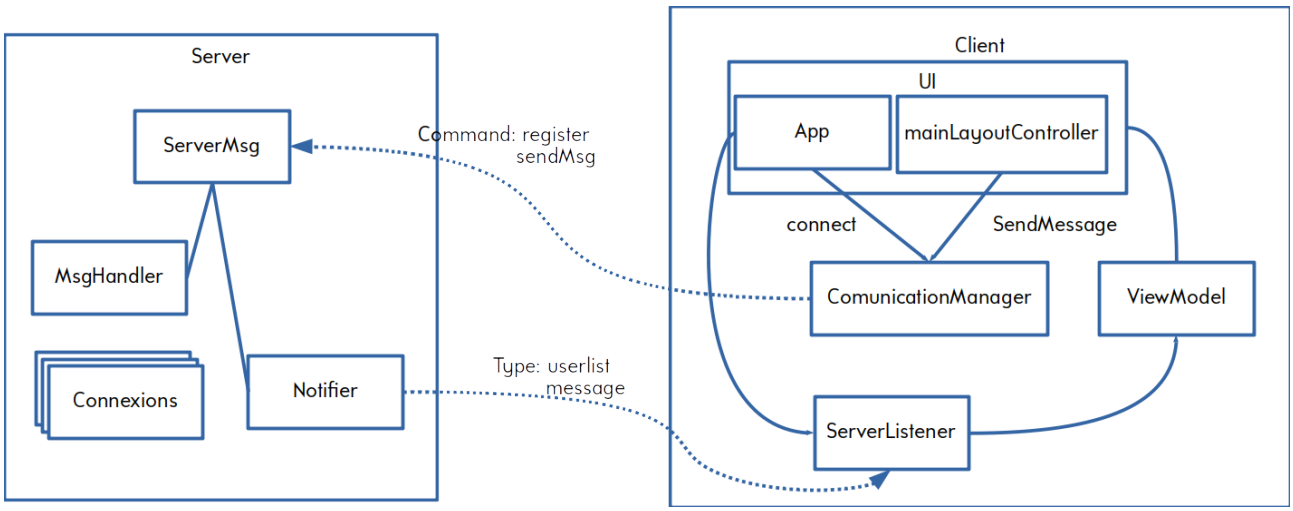
Quan el client s'inicia, ha de registrar-se al servidor (que evidentment ha d'estar en funcionament). Amb aquest registre, el servidor coneixerà tots els clients que té connectats. Quan un client envia un missatge al servidor, aquest el reenvía a tots els clients que té registrats. En cas que l'enviament a algun client no es produisca, s'entén que el client s'ha desconnectat, i el treu de la llista.

El servidor envía també periòdicament la llista d'usuaris als clients, actualitzada.

Aquest model es coneix com patró observer, que es basa en aquest mecanisme de subscripció. El client el subscriu, i el servidor envía les publicacions als subscriptors. Teniu més informació a Refactoring Guru.

Açò suposa una dificultat afegida al client, ja que aquest, haurà d'implementar també un servei per tal de rebre les notificacions. Així, el client crearà aquest servidor en un port aleatori, i quan es connecte al servidor li indicarà a quin port li ha d'enviar les notificacions. Fent un símil amb una subcripció a una revista en paper, caldría indicar quina és l'adreça a la que volem rebre les publicacions.

L'esquema general, de forma gràfica serà el següent:



1.2 Estructura de l'aplicació

Com hem dit, l'aplicació consta del mòdul client i el mòdul servidor.

Veiem-ne l'estructura:

1.2.1 Estructura del mòdul servidor

```
server/
|-- build.gradle
`-- src
    `-- main
        |-- java
        |   `-- com
        |       `-- ieseljust
        |           `-- psp
        |               `-- server
        |                   |-- Connexio.java
        |                   |-- MsgHandler.java
        |                   |-- Notifier.java
        |                   `-- ServerMsg.java
        `-- resources
```

Com veiem, es treballa amb el paquet `com.ieseljust.psp.server`, i aquest conté les següents classes:

- Classe `ServerMsg` **[A completar!]**: Implementa el servidor de missatges com a tal. Escoltarà pel port 9999 i atendrà peticions que li arriben per part dels clients. A més periòdicament enviarà notificacions als clients registrats (ho expliquem més avall).
- Classe `MsgHandler` **[A completar!]**: Serà la classe que gestionarà les peticions en sí. Quan la classe `ServerMsg` reba una petició de servei, qui la gestionarà en un fil a banda serà aquesta classe. Haurà de descodificar el missatge (que rebem en mode text) segons el protocol (format JSON), i determinar l'acció a realitzar segons s'indique al missatge.
- Classe `Connexio`: Representa cadascuna de les connexions dels clients. Conté:
  - `private String user`: el nom d'usuari,
  - `private Socket socket`: el socket pel que s'ha connectat el client al servidor,
  - `private int listenPort`: El port pel qual **el client** va a rebre les notificacions del servidor (consulteu l'esquema per entendre-ho millor)

  A més, aquesta classe proporciona getters i setters per a les diferents propietats, i el que més ens interessa, proporciona els mètodes:

  - `public String getRemoteAddress()`: Que retorna l'adreça IP del client connectat, obtinguta a través del socket.
  - `public int getRemotePort()`: Que retorna el port pel que el client connectat espera rebre les notificacions.

1.2.2 Estructura del mòdul client

```
client
|-- build.gradle
`-- src
    `-- main
        |-- java
        |   `-- com
        |       `-- ieseljust
        |           `-- psp
        |               `-- client
        |                   |-- App.java
        |                   |-- communications
        |                   |   |-- communicationManagerException.java
        |                   |   |-- communicationManager.java
        |                   |   `-- serverListener.java
```

```
|                   |-- CurrentConfig.java
|                   |-- itemViewFactory.java
|                   |-- itemView.java
|                   |-- mainLayoutController.java
|                   |-- Message.java
|                   `-- ViewModel.java
`-- resources
    |-- background.jpg
    |-- itemView.fxml
    |-- mainLayout.fxml
    `-- messageList.css
```

La carpeta `resources` conté els diferents recursos necessaris per a la interfície: el fons, un fitxer CSS amb alguns estils, i dos fitxers fxml amb els *layouts* de la interfície. Aquests layouts es carregaran des de l'aplicació principal. **No serà necessari modificar res d'aquests fitxers**.

El paquet principal és `com.ieseljust.psp.client`, i conté les següents classes:

- App: Es tracta de la classe principal. Com podreu vore, deriva de `javafx.application.Application`, el que vol dir que és una aplicació JavaFX. Aquesta classe s'encarrega de crear la finestra de l'aplicació i injectar en ella el fitxer *fxml* amb la interfície. A més, crea un fil dins la pròpia interfície (podeu vore com ho fa per vore com llançar un runnable sense crear la classe). Aquest fil, periòdicament actualitzarà la llista d'usuaris i de missatges a partir de les dades la classe *ViewModel*.
- `mainLayoutController`: És el controlador de la vista principal. S'encarrega de gestionar els esdeveniments de la interfície, com enviar un missastge quan es fa clic al botó d'enviar, i mantindre les llistes d'usuaris i missatges sincronitzades (com els observables) amb el model. En principi, tampoc caldrà implementar res aci.
- `ViewModel`: S'encarrega d'emmagatzemar les dades necessàries per a les vistes: la llista d'usuaris i de missatges. En ella escriu la classe App, amb la informació actualitzada, i d'ella llegirà el mainLayoutController per actualitzar la interfície. Tampoc haurem de modificar res d'aquest aclasse.
- `Message`: Representa un missatge del xat. Conté dos propietats amb l'usuari i el missatge. A més dels getters i els setters, ofereix els mètodes `toJSON()`, que obté una representació del missatge en JSON, i `toJSONCommand()`, que obté el missatge en JSON amb un component més `"command":"newMessage"`, preparat ja per enviar al servidor.
- `CurrentConfigJava`: Conté la configuració de l'aplicació.
- `itemView` i `itemViewFactory`: S'encarreguen de gestionar a la interfície la visualització dels missatges.

En aquest paquet, tenim el subpaquet `communications`, que és el paquet que huareu de completar amb funcionalitat. Aquest subpaquet conté:

- Classe `communicationManagerException`: Implementa una excepció personalitzada de comunicacions de l'aplicació. S'utilitzarà per llançar una excepció quan l'usuari ja està registrat.
- Classe `communicationManager` **[A implementar!]**: S'encarrega de la gestió de les comunicacions amb el servidor. Caldrà implementar els tres mètodes següents:
  - `public static JSONObject sendServer(String msg)`: Enviarà un string al servidor que contindrà un missatge JSON i gestionarà la resposta d'aquest. Aquest mètode serà de suport als dos mètodes següents.
  - `public static void connect() throws JSONException, communicationManagerException`: Crea un misstge pe al servidor amb l'ordre register, el nom d'usuari i el port pel qual el client escoltarà les notificacions (el teniu a CurrentConfig.listenPort()). Aquest missatge l'enviarà al servidor a través de sendServer, i gestionarà els errors que puga donar.
  - `public static void sendMessage(Message m)`: Envía el missatge al servidor, indicant l'ordre `newMsg`.
- Classe `serverListener implements Runnable` **[A completar!!]**: La classe Aplicació inicia un fil amb aquesta classe (necessita un fil a banda del principal, ja que el principal gestiona la interfície). Aquest fil, crearà un socket (de tipus servidor) en un port aleatori (ja ho teniu implementat), i el guardarà a `listenPort`, en la configuració general. El mètode *run* d'aquesta classe farà el següent:
  - 1. Crear un socket de tipus servidor que escolte pel port de recepció de missatges (ja implementat, però és interessant que consulteu com està fet)
  - 2. Inicea un bucle infinit a l'espera de rebre notificacions per part del servidor. Quan arribe una notificació la processarem de manera adeqüada. Les peticions que podme rebre seran de tipus:
    * `{"type": "userlist", "content": [Llista d'usuaris]}`: amb un JSONArray amb la llista d'usuaris.
    * `{"type": "message", "user":"usuari", "content": "Contingut del missatge" }`: Amb els missatges.

    És interessant implementar un mètode a banda per processat aquestes notificacions, però no cal que creem un fil a propòsit per atendre cada missatge, ja que no som un servidor com a tal, i el que estem fent aci, és mantindre un canal de recepció només amb el servidor.

Una vegada obtingueu la informació de les notificacions, caldrà actualitzar el ViewModel amb aquesta informació. Per a això, s'utilitza la propietat `ViewModel vm` que és inicialitzada en el constructor d'aquesta classe. Les actualitzacions que caldrà fer són:

- Per actualitzar la llista d'usuaris: `vm.updateUserList(users);`, sent `users` un `ArrayList<String>` format pels usuaris.
- Per enviar un missatge: `vm.addMessage(msg);`, sent msg un objecte de tipus `Message`.