

## Introduction

Le but de ce TP qui s'étalera sur l'ensemble des séances restantes est de vous faire travailler en binôme sur un système de client/serveur de « chat » :

1. Un serveur de chat est lancé et attends des connections des différents clients.
  2. Un ou plusieurs clients de chat se connectent au serveur et lui envoient des messages, le serveur rediffuse alors les messages de chaque client à tous les autres clients.
    - a. Le client peut être un client simple fonctionnant dans la console (Client 1 sur la Figure 1).
    - b. Le client peut aussi utiliser une interface graphique (comme le Client 2 dans la Figure 1).
- Un client se déloge du serveur en lui envoyant le message « bye ».

Vous pourrez trouver une ébauche du code à réaliser dans l'archive : /pub/ILO/TP5.zip

Documentation Java : [http://docs.oracle.com/javase/\[7 ou 8\]/docs/api/](http://docs.oracle.com/javase/[7 ou 8]/docs/api/)

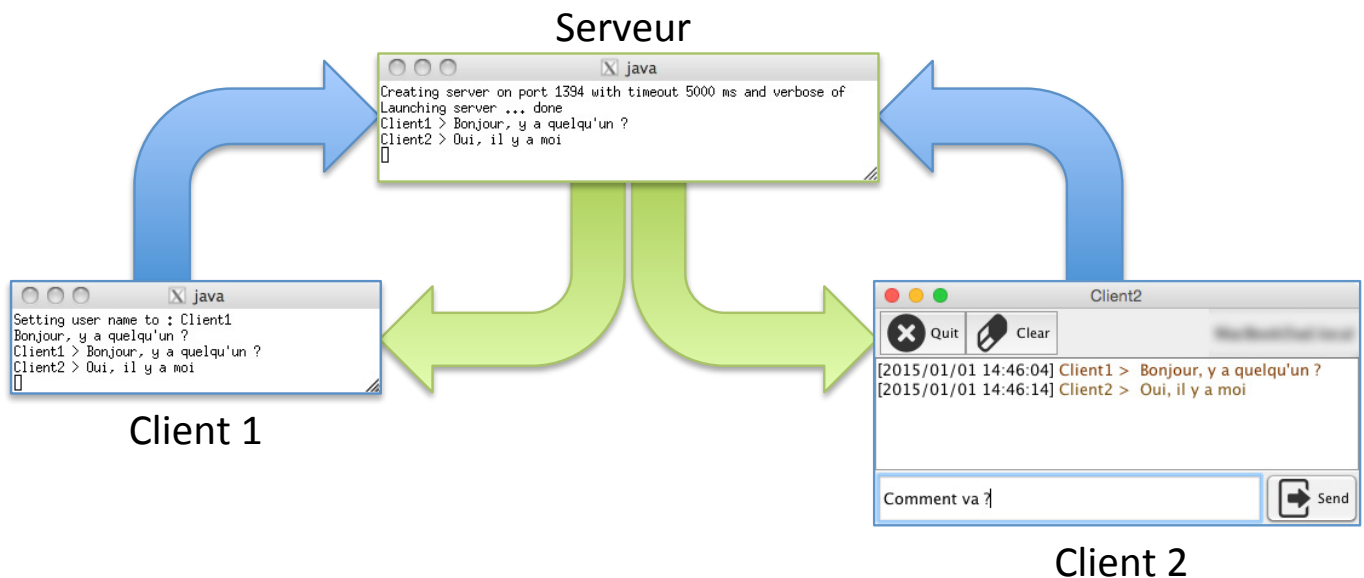


Figure 1 : clients / serveur de "chat"

## Travail à réaliser

### 1) Etude de l'existant

Votre première tâche sera d'étudier le code qui vous est fourni afin de comprendre son fonctionnement.

Une clé principale pour comprendre le fonctionnement de ce système de client / serveur sont les flux d'entrée / sortie (**{Input|Output}Stream**) qui sont encapsulés dans les sockets utilisées pour communiquer entre les clients et le serveur :

Le serveur crée une `ServerSocket` :

```
serverSocket = new ServerSocket(port);
```

Puis attends qu'un client se connecte :

```
Socket clientSocket;
```

...

```
clientSocket = serverSocket.accept();
```

Une fois le client accepté, le serveur a alors accès au flux d'entrée en provenance du client :

```
clientSocket.getInputStream(); à partir duquel on peut créer un
```

`new BufferedReader(new InputStreamReader(...));` pour lire du texte en provenance du client.

Le serveur récupère de la même manière le flux de sortie vers le client :

`clientSocket.getOutputStream();` à partir duquel on peut créer un `new ObjectOutputStream(<OutputStream>);` pour envoyer des messages vers le client. Les messages sont ici représentés par des objets de type « Message » contenant une date, un contenu et éventuellement un auteur.

A chaque nouvelle connexion d'un client, celui-ci est traité dans un nouveau thread afin que plusieurs clients puissent se connecter au serveur. Cette tâche est effectuée par un « ClientHandler » Runnable dont le travail (la méthode run) consiste à lire le flux d'entrée texte en provenance du client, puis lorsqu'une ligne est lue, à la traiter en composant un « Message » à l'ensemble des clients du serveur.

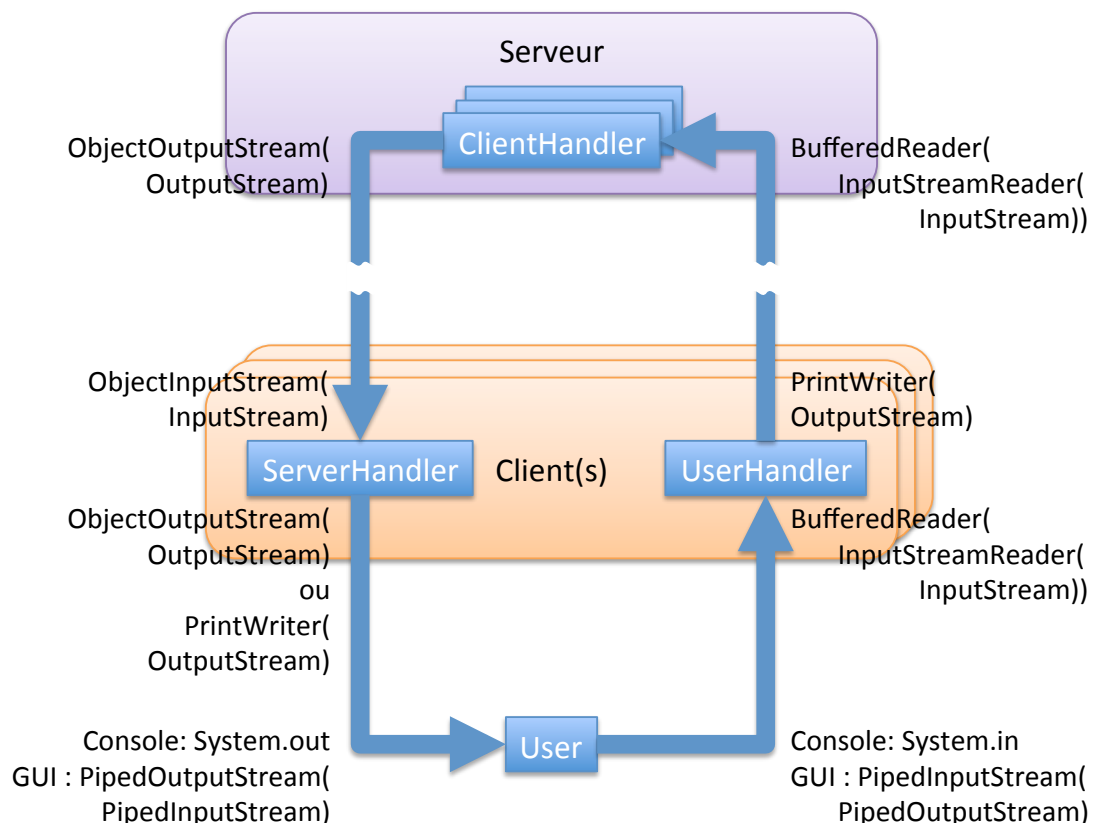
Le client « ChatClient » quant à lui crée une socket :

`clientSocket = new Socket(host, port);`

Puis récupère les flux d'entrée / sortie depuis / vers le serveur ainsi que les flux vers / depuis l'utilisateur.

Le « ChatClient » gère les flux d'entrée/sortie obtenus du serveur au travers de la « clientSocket » ainsi que les flux d'entrée/sortie de l'utilisateur au travers de deux classes spécifiques :

- « ServerHandler » Runnable dont le travail consiste à lire le flux objet de « Message » diffusés par le serveur sur le flux d'entrée de la socket et à envoyer les messages vers le flux de sortie de l'utilisateur (le System.out dans le cas d'un client console).
- « UserHandler » Runnable dont le travail consiste à lire le flux d'entrée texte en provenance de l'utilisateur (le System.in dans le cas d'un client dans la console) et à envoyer ce texte au serveur grâce au flux de sortie de la socket.



**Figure 2 : Nature des flux d'IO**

Le schéma de la Figure 2 résume la nature des flux d'entrées et de sorties utilisés dans cette architecture client / serveur.

La seule différence entre un client « console » et un client « graphique » du point de vue du « ChatClient » se situera dans les flux d'entrée/sortie de l'utilisateur :

- Dans le cas du client console on utilisera :

- Le System.out (un PrintStream→FilterOutputStream→OutputStream) pour l’affichage des messages du serveur dans la console.
- Le System.in (un InputStream) pour la lecture des messages à envoyer au serveur depuis la console.
- Dans le cas du client graphique on utilisera (voir Figure 4 et Figure 5, page 5) :
  - Un PipedOutputStream→OutputStream connecté à un PipedInputStream dans la fenêtre graphique pour afficher les messages dans un JTextPane (une zone de texte permettant d’afficher du texte riche sur plusieurs lignes) ou plus exactement dans le StyledDocument qui contient le texte riche du JTextPane.
  - Un PipedInputStream→InputStream connecté à un PipedOutputStream dans la fenêtre graphique pour lire les messages à envoyer au serveur depuis JTextField (zone de texte simple, éditable ne contenant qu’une ligne de texte).

## 2) UML

Etudiez le graphe de classes ci-dessous (Figure 3) qui illustre les principales classes utilisées dans l’architecture client/serveur du « chat » ainsi que le code source.

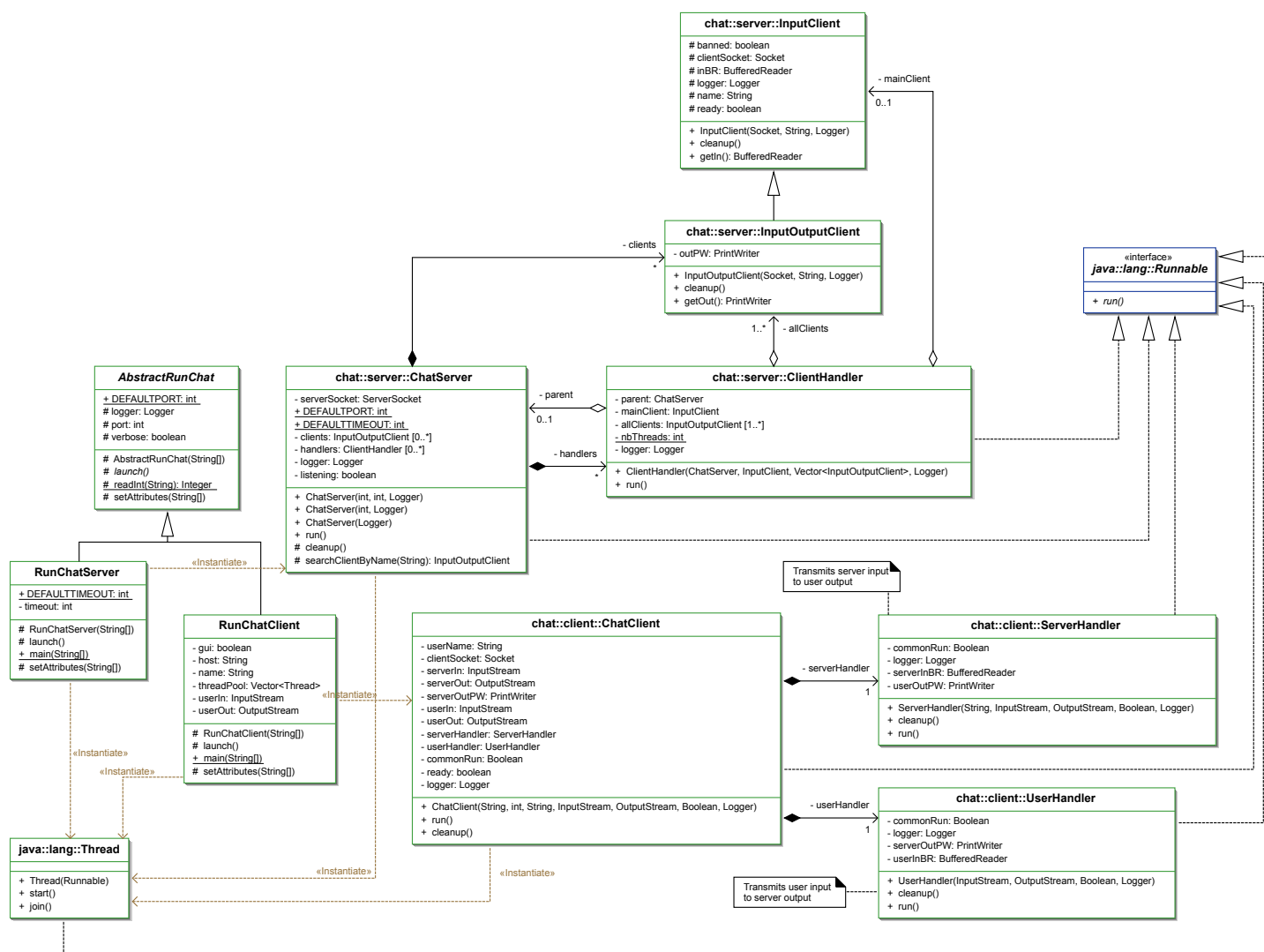


Figure 3 : Classes utilisées dans l'architecture client/serveur

Après avoir étudié le code et le graphe de classes, répondez (individuellement) à ces quelques questions<sup>1</sup> :

- A quoi sert la classe `AbstractRunChat` ?
- Expliquez la relation `ChatServer / InputOutputClient` concrétisée par l'attribut « clients ».
- Expliquez la relation `ClientHandler / InputClient` concrétisée par l'attribut « mainClient ».
- Expliquez la relation `ClientHandler / InputOutputClient` concrétisée par l'attribut « allClients ».
- Combien de threads tournent sur un serveur dans le scénario présenté par la Figure 1 (page 1) ?  
Détaillez votre réponse en précisant qui lance qui.
- Combien de threads tournent dans le Client 1 du scénario présenté par la Figure 1 (page 1) ?  
Détaillez votre réponse en précisant qui lance qui.
- A quoi sert le `threads[i].join()` à la fin du run de `ChatClient` ?
- Que représente la classe `ChatClient` dans le cadre d'une architecture MVC ?

Vous pourrez écrire vos réponses dans le fichier « reponses.txt » à la racine du projet et envoyer le fichier à votre chargé de TD avant la fin des deux premières séances de TD.

### 3) Complétion du Client

Complétez les classes `ChatClient`, `UserHandler` et `ServerHandler` en suivant les « TODO » afin d'obtenir un moteur de client de « chat » en état de fonctionner.

Il s'agira pour la classe « `ChatClient` » de :

1. ⇒ Créer la socket de connexion au serveur
2. ⇒ D'obtenir le flux de sortie vers le serveur pour lui envoyer les messages du client.
3. ⇒ D'obtenir le flux d'entrée depuis le serveur pour lire les messages qu'il rediffuse à tous les clients.

Pour la classe `UserHandler` il s'agira de :

1. ⇒ Créer un `BufferedReader` pour lire les lignes tapées par l'utilisateur (soit dans la console soit dans un `TextField`)
2. ⇒ Créer un `PrintWriter` pour envoyer les messages tapés par l'utilisateur au serveur
3. ⇒ Compléter la méthode `run` en :
  - a. Lisant les entrées de l'utilisateur avec le `BufferedReader`
  - b. Envoyant les entrées utilisateur au serveur en utilisant le `PrintWriter`
  - c. Quittant la boucle principale du run si le client a tapé la command de déconnection « bye ».

Pour la classe `ServerHandler` il s'agira de :

1. ⇒ Créer un `ObjectInputStream` pour lire les « Message »s envoyés par le serveur
2. Créer un soit
  - a. ⇒ Un `PrintWriter` pour envoyer les messages envoyés par le serveur sous forme de texte à l'utilisateur.
  - b. ⇒ Un `ObjectOutputStream` pour envoyer les messages sous forme d'objets à l'utilisateur.
3. Compléter la méthode `run` en :
  - a. ⇒ Lisant les entrées du serveur avec l'`ObjectInputStream`.
  - b. ⇒ Envoyant les entrées du serveur à l'utilisateur en utilisant soit le `PrintWriter`, soit l'`ObjectOutputStream`.

---

<sup>1</sup> Vous pourrez profiter de la recherche de références à un élément sélectionné dans eclipse pour trouver rapidement les réponses à ces questions : bouton droit → References → Project.

#### 4) Etude du premier client graphique

Vous avez à votre disposition un premier client graphique que vous pourrez étudier pour vous en inspirer dans le nouveau client graphique.

Le premier client graphique se compose d'une JFrame dans laquelle sont placés les différents widgets nécessaires, la Figure 4 présente l'aspect du premier client graphique et la Figure 5 sa structure :

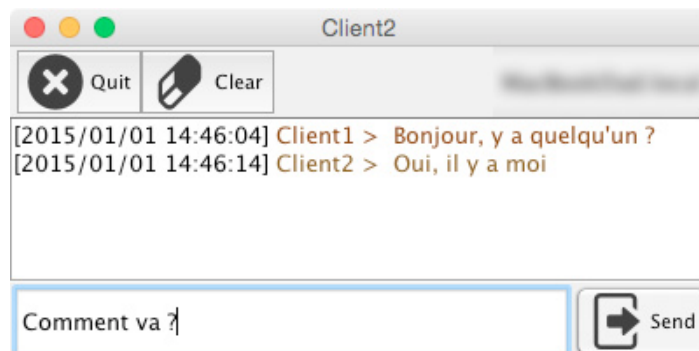


Figure 4 : Client graphique

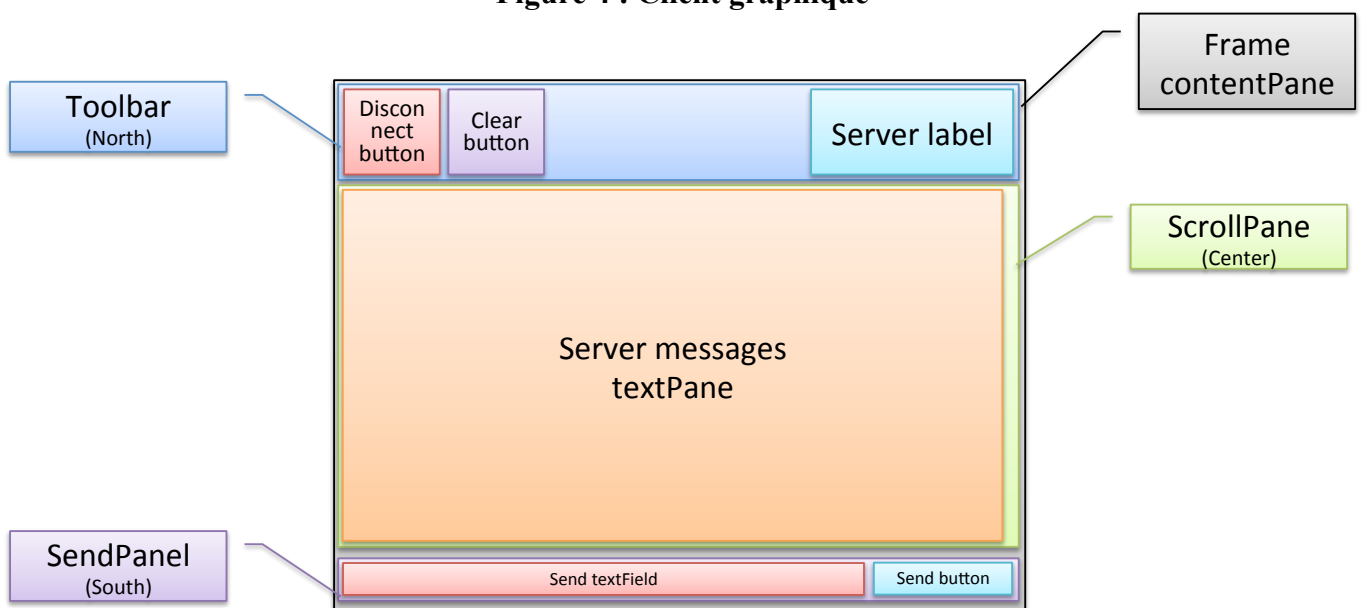


Figure 5 : Structure du client graphique

Les fonctionnalités du client graphique sont les suivantes (à chaque fonctionnalité correspond une « Action » qui permet d'implémenter cette fonctionnalité) :

- **SendAction** : Action à réalisée pour envoyer le texte tapé par l'utilisateur dans le « Send textField » au serveur.
  - Cette action est associée au « Send button » ainsi qu'au « Send textField »
- **ClearAction** : Action à réalisée pour effacer les messages écrits dans le « Server messages textPane ».
  - Cette action est associée au « Clear button » de la « Toolbar ».
- **QuitAction** : Action réalisée lorsque l'utilisateur veut se déconnecter du serveur et ainsi quitter l'application (On envisagera pas les cas de déconnexion / reconnexion).
  - Cette action est associée au « Disconnect button » de la « Toolbar ». On peut aussi la réutiliser lorsque l'on ferme la fenêtre de l'application pour se déconnecter proprement du serveur de « chat ».
- Ces trois actions sont aussi regroupées dans un menu « Actions ».

Les widgets du client graphique sont les suivants :

- Le textPane affiche les messages du serveur.
  - Les messages à afficher sont précédés de la date de réception du message.

- Les messages proprement dits sont affichés dans une couleur générée à partir d'un entier aléatoire dont le générateur est initialisé avec le hashCode du nom d'utilisateur afin de toujours obtenir la même couleur pour un même utilisateur.
- Les messages dont la forme est « **utilisateur** > **message** » sont donc parsés à la recherche du nom d'utilisateur.
- Le Send textField permet de taper un message à envoyer au serveur lorsque l'on tape entrée.
- Le Send button permet d'envoyer le message du Send textField au serveur.
- Le bouton de déconnexion déconnecte le client du serveur et quitte l'application.
- Le bouton clear efface le contenu du textPane affichant les messages du serveur.

En dehors de son aspect graphique le client graphique doit aussi écouter les messages qui lui seront retransmis par le « ServerHandler » du « ChatClient ». Il faudra donc aussi compléter sa méthode run (ainsi que diverses méthodes utilitaires) pour qu'il lise en permanence les messages en provenance du serveur et les affiche au fur et à mesure de leur réception dans le « StyleDocument » contenu dans le textPane. Le client graphique utilise des Piped{Input|Output}Stream afin de se connecter aux {Output|Input}Stream des « UserHandler » et « ServerHandler » du « ChatClient » ce qui permet de rediriger ces flux dans des Piped{Output|Input}Stream situés dans la fenêtre graphique.

⇒ Complétez la méthode run de la classe « ClientFrame » ainsi que les méthodes utilitaires qu'elle utilise.

⇒ Complétez la méthode launch de la classe RunChatClient afin d'obtenir les flux d'entrée/sortie de la « ClientFrame » et les fournir ainsi au « ChatClient ».

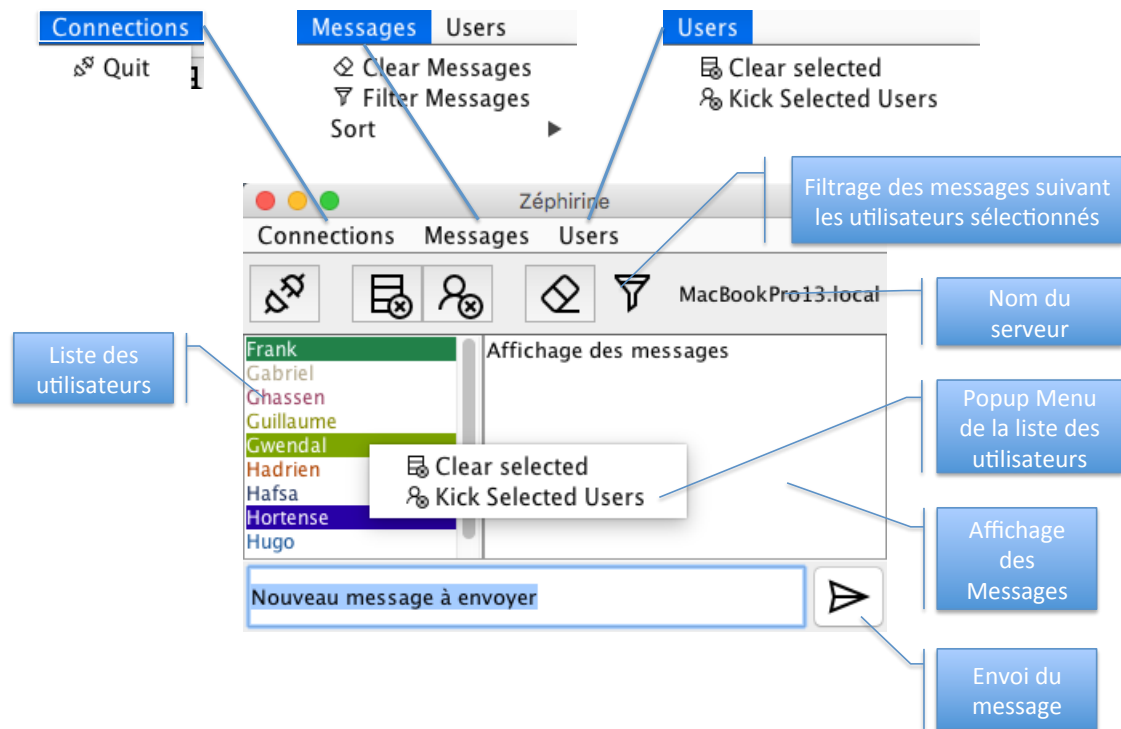
### 5) Création d'un nouveau client graphique

Vous pourrez pour construire ce nouveau client graphique vous inspirer des classes de l'exemple précédent ainsi que des « ExampleFrame » et « ListExampleFrame » que vous pourrez ouvrir avec le « WindowBuilder Editor » et lancer avec les classes « RunExampleFrame » et « RunListFrame ».

Vous allez maintenant créer une nouvelle fenêtre en tant que client graphique de chat « ClientFrame2 » ayant les caractéristiques suivantes :

- Le flux des données en sortie du ChatClient est un flux d'objets de type « Message » et non plus un flux texte afin de transmettre directement les « Message »s au client graphique.
- Le client graphique maintient (au travers d'une classe héritière d'AbstractListModel<String>) et affiche (au travers d'une JList<String>) une **liste d'utilisateurs uniques triée par ordre alphabétique** des utilisateurs ayant envoyé un message. Cette liste sera utilisée pour filtrer la liste des messages en fonction des éléments sélectionnés dans cette liste. Par ailleurs, cette JList affiche les noms d'utilisateurs colorés tels qu'ils l'étaient dans « ClientFrame » grâce à une classe héritière de ListCellRenderer<String> comme dans l'exemple ListExampleFrame.java.
- Le client graphique conserve l'ensemble des messages dans une **liste des messages** afin de pouvoir réaliser sur cette liste les opérations suivantes :
  - Filtrage des messages en fonction des utilisateurs sélectionnés dans la liste d'utilisateurs.
  - Tri des messages suivant différents critères :
    - Tri par date (ordre naturel).
    - Tri par contenu.
    - Tri par auteur.
- Le client graphique affiche les messages colorisés (comme dans la première version du client graphique) :
  - Soit de la même manière que le premier client graphique : Dans un JTextPane.
  - Soit dans une JList.
  - Soit dans une JTable ce qui peut permettre de séparer la date, l'auteur et le contenu.

Le nouveau client graphique doit avoir à peu près l'aspect présenté dans la Figure 6.



**Figure 6 : Aspect du nouveau client graphique**

## Echéancier

Ce TP s'étalera sur les 7 dernières séances de TD.

- A la fin des deux premières séances (d'1h45) envoyez (individuellement) par mail votre fichier « reponses.txt » à votre chargé de TD.
- A la fin de la 4<sup>ème</sup> séance de TD (ou même avant si vous êtes prêts), nous noterons la partie 3 (Client console). Vous devrez donc terminer la partie 3 avant la fin de cette séance de TD.
- Pour le rendu final nous noterons la partie 5 (Client graphique). Vous déposerez alors une archive de votre projet sur le serveur de dépôt de projets.