haute école
neuchâtel berne jura
arc ingénierie
www.he-arc.ch

TP2 Java : Chat JavaFX

Rapport technique

À l'attention de M. Carrino

Java –TP2
Fleury Anthony, Guerne Jonathan
Mai 2017

1 Introduction

Le but de ce projet est de **réaliser une application de Chat en Java** permettant à un ou plusieurs clients de communiquer entre eux au travers du réseau (local). Le chat doit avoir **un seul canal** contenant tous les messages, il n'y a donc pas de notion de messages privés entre clients.

Deux types de messages peuvent être transmis (év. un troisième en bonus) :

- Envoie de message texte classique
- Communication vocale (sous forme d'appel et pas d'enregistrements)
- Envoie de fichiers (**Bonus**)

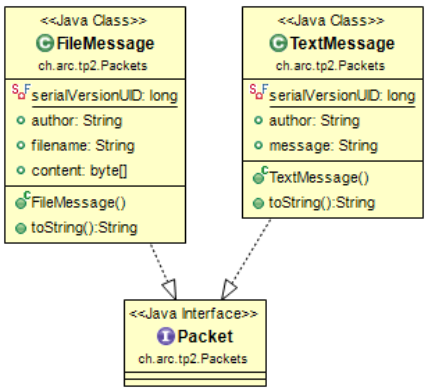
L'application doit être réalisée en JavaFX.

2 Fonctionnement de l'application

Notre implémentation possède deux parties distinctes : la **partie client** et la **partie server**. Un client s'enregistre auprès d'un serveur afin de pouvoir recevoir les différents **Messages** échangés sur le chat. Plusieurs clients peuvent être connectés en simultané au même serveur.

L'application client est une application graphique, mais le serveur est un simple programme en ligne de commande avec des sorties console.

2.1 Packets



Dans le but d'obtenir plus d'informations qu'un simple *String* nous avons choisi de **transférer des objets entre le serveur et les clients**. Afin d'organiser les objets envoyés nous avons créé une interface *Packet* sérialisable dont tous les objets à envoyer hériteront.

Les implémentations concrètes de *Packet* dans notre projet sont *TextMessage* pour envoyer un message dans le chat en précisant l'auteur et *FileMessage* pour transmettre un fichier.

On peut remarquer qu'il n'y a pas de *Packet* lié à la transmission de données audio. Malheureusement, par manque de temps nous n'avons pas eu le temps d'implémenter cette fonctionnalité (plus de détail dans la conclusion).

2.2 Serveur

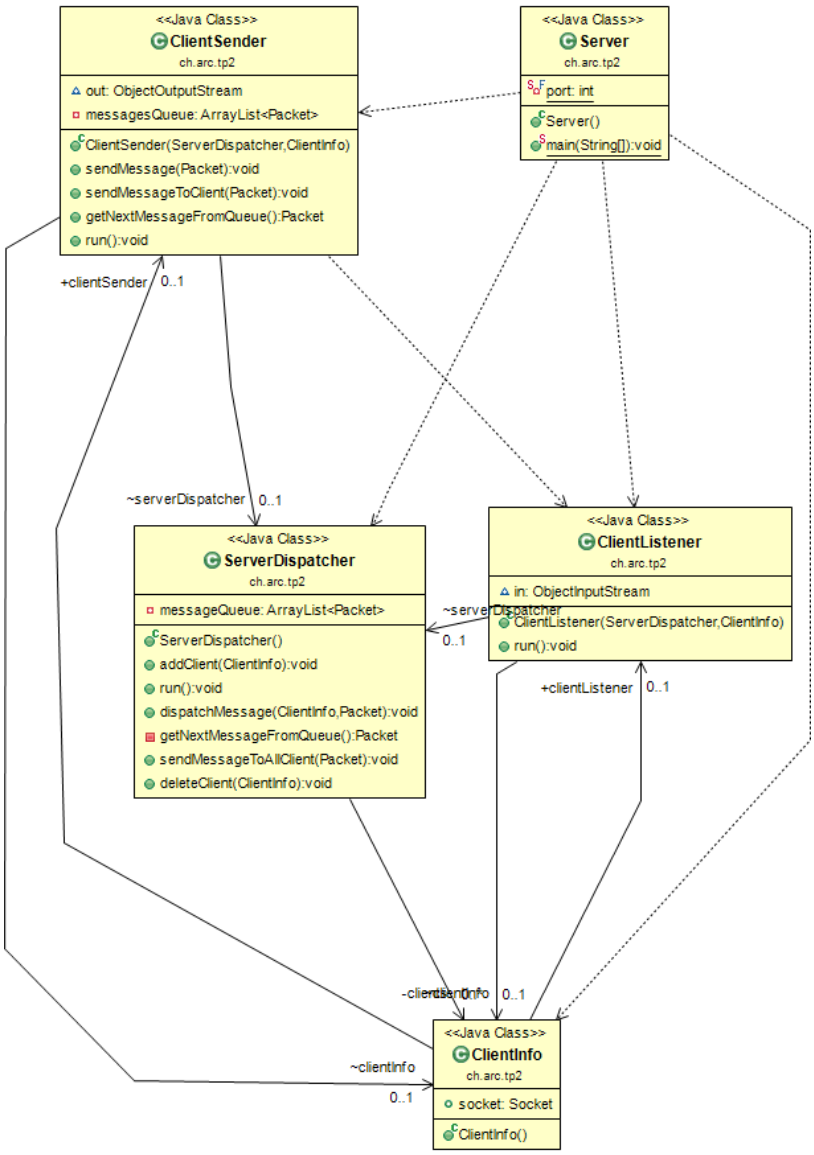


Diagramme de classe du server

De manière à gérer plusieurs clients, le serveur utilise un *ServerDispatcher* qui va permettre de gérer les envois en broadcast à tous ses clients.

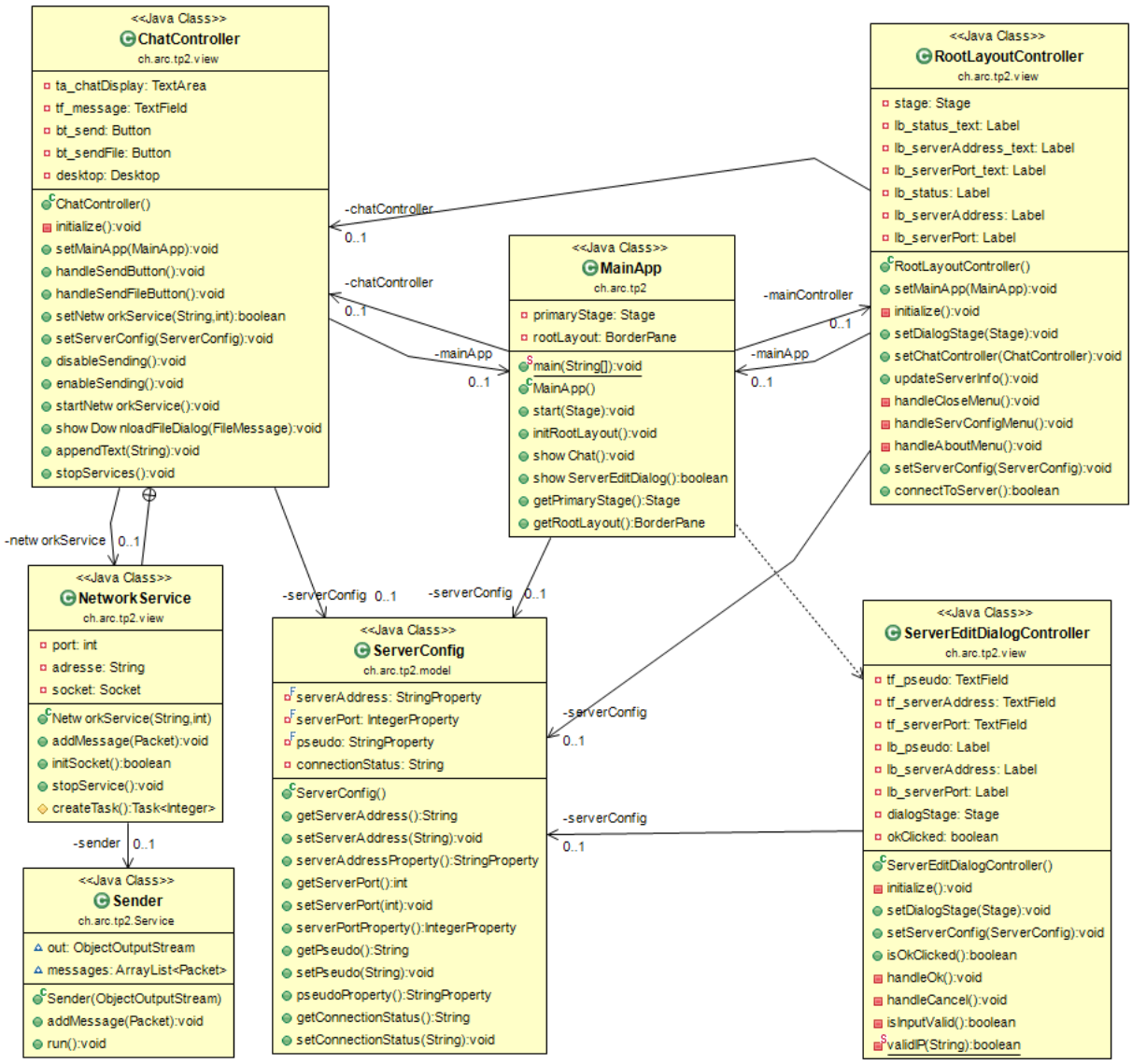
Chaque client possède un *socket*, un *clientListener* et un *clientSender*. Ces éléments sont liés entre eux par la classe *ClientInfo*.

ClientSender et *ClientListener* permettent respectivement d'envoyer des *Packets* à un client précis et d'écouter les *Packets* émanant d'un client précis.

Après avoir reçu un message, *ClientListener* le transmet au *ServerDispatcher* qui va lui se charger de transmettre ce message en broadcast à tous les clients (même à l'émetteur original).

À noter que le diagramme de classe de *Packet* devrait être présent ici puisqu'il est utilisé par les classes *ServerDispatcher*, *ClientSender* et *ClientListener*. Il n'a pas été rajouté par soucis de lisibilité, la même problématique est apparue pour le diagramme de classe de la partie client.

2.3 Client



Le client est une application JavaFX qui va permettre d'afficher le chat, mais également de configurer l'adresse et le port du serveur. Il utilise le modèle MVC pour transmettre les données de configurations vers l'affichage.

L'affichage principal se fait dans un *RootLayoutController*. Ce dernier possède un *ChatController* qui va se charger d'afficher le chat (*textArea* pour les messages reçus, *textfiield* pour l'envoi ainsi que quelques boutons). Lors de la configuration du serveur on utilise *ServerEditDialogController* pour modifier le pseudo de l'utilisateur sur ce serveur, l'adresse ou encore le port du serveur.

La class *NetworkService* est une classe interne à *ChatController*, elle va permet d'initier la connexion au serveur et pourra ensuite être utilisée comme un service en arrière-plan pour recevoir les différents *Packets* émanant du serveur. L'envoi se fait lui avec la classe *Sender* qui sera instanciée par le *NetworkService*.

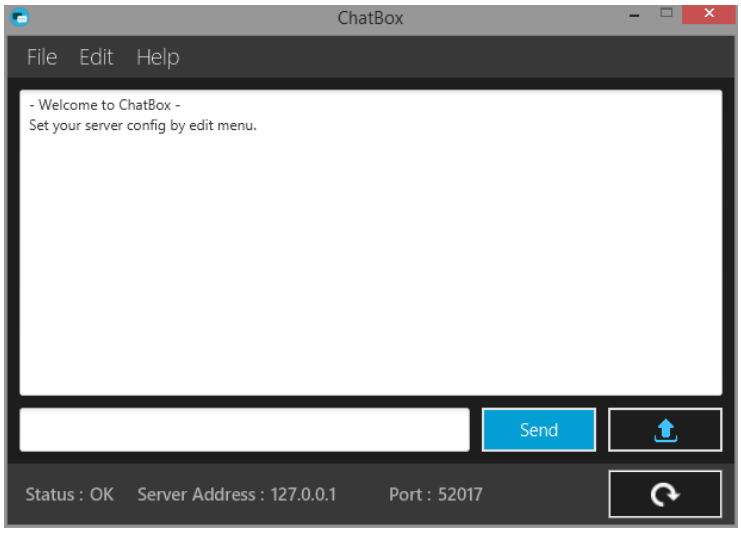
3 Choix Architectural

Nous avions opté dès le début du projet pour un chat permettant la connexion de plusieurs clients sur le même serveur. Le serveur a été inspiré et réadapté d'un exemple trouvé en ligne à l'adresse suivante : http://inetjava.sourceforge.net/lectures/part1_sockets/InetJava-1.9-Chat-Client-Server-Example.html

L'utilisation du *ServerDispatcher* permet de facilement gérer le transit des messages entre les clients. L'emploi du polymorphisme pour les objets transmis (tous sont des *Packets*) nous a permis de travailler exclusivement avec la classe parent dans le serveur sans porter d'intérêt à quelle classe concrète avait été transmise. Cela a rendu les modifications plus simples et permettra de facilement implémenter l'ajout de nouveaux types de *Packets*.

Le modèle MVC utilisé pour le client permet également d'avoir un squelette d'application plus solide, qui lui aussi, permettra d'aisément implémenter de nouvelle fonctionnalité par la suite tout en restant propre.

4 Déploiement et utilisation



Screenshot du programme de chat (client)

Pour que le programme de chat puisse fonctionner, il est impératif que le serveur soit en état de marche sur une machine. Celui-ci peut être lancé sur n'importe quelle machine, il faut toutefois être attentif au fait que les clients ainsi que le serveur doivent utiliser le même couple adresse-port de manière à pouvoir communiquer.

Une fois l'application lancée, le programme tente de se connecter au serveur *localhost* (127.0.0.1) avec le port 52017.

L'utilisateur peut éditer ces informations en ouvrant la fenêtre de configuration sous « *Edit* > *Edit Server Configuration* ».

Pour envoyer un message il suffit d'écrire dans le champ de saisie situé en bas de la fenêtre et d'appuyer sur le bouton *Send* pour lancer l'envoi (le touche Entrée fonctionne également).

L'envoi de fichier se fait en cliquant sur le bouton avec le logo upload (à côté de *Send*). Les fichiers transférés le sont à tous les utilisateurs et sont sauvegardés dans un dossier nommé *ChatDownloads* situé au sein du répertoire utilisateur par défaut du système d'exploitation. Une boîte de dialogue ouverte lors de la fermeture de l'application demandera à l'utilisateur s'il souhaite effectuer un nettoyage du dossier des éléments transférés ou non.

Enfin, le bouton de rafraîchissement positionné au bas de la fenêtre permet de retenter une connexion au serveur avec les paramètres actuels.

5 Conclusion

Nous sommes globalement satisfaits du programme. Il est toutefois dommage de ne pas avoir pu implémenter la gestion d'un chat audio, mais la difficulté d'implémentation additionnée à notre emploi du temps chargé nous ont encouragés à laisser tomber ce point au profit du transfert de fichier. Ce dernier s'est avérer plus simple à mettre en place.

Le projet pourrait être plus approfondit en implémentant, comme dit plus haut, la gestion de l'audio ou encore en ajoutant des fonctionnalités liées à la réception de fichier (lien pour retrouver les fichiers dans le chat, pop-up pour télécharger ou non le fichier).