Serveur et client ChatFusion

Programmation réseaux

Irwin Madet & Nicolas Van Heusden

8 mai 2022

Table des matières

1 Introduction

Ce rapport détaille le fonctionnement et l'élaboration d'un serveur ainsi qu'un client implémentant le protocole ChatFusion. Ce protocole repose sur TCP/IP, et permet une communication textuelle en temps réel. Le protocole permet également l'envoi de messages privés, le transfert de fichiers en privé, ainsi que la fusion de plusieurs serveurs pour ne former qu'un seul réseau homogène.

N.B : Pour de plus amples détails sur le fonctionnement du protocole, veuillez vous référer à la RFC.

2 Utilisation

2.1 Serveur

Le serveur se présente sous la forme d'un JAR exécutable. Il a besoin de plusieurs informations pour démarrer :

1. Un identifiant sur un entier 32 bits, utilisé pour l'identification sur le réseau ChatFusion ;
2. Un port sur lequel écouter les connexions entrantes.

Il est important de notifier que le serveur démarrera avec n'importe quel identifiant, cependant la fusion peut échouer si un autre serveur possède le même identifiant. Il revient à l'administrateur du réseau de choisir des identifiants distincts pour tous ses serveurs.

Le serveur propose également un système de fusion, via la commande `fusion <adresse> <port>`. Cette commande va lancer un processus de fusion avec le serveur à l'adresse donnée. Lorsque deux serveurs ou plus sont fusionnés pour ne former qu'un seul réseau, les paquets transitent à travers les différents serveurs, ainsi les messages publics sont visibles pour tous les utilisateurs, indépendamment du serveur sur lequel ils sont connectés. Les messages privés et transferts de fichiers sont également disponibles, à condition de préciser le serveur de destination.

2.2 Client

Tout comme le serveur, le client se présente sous la forme d'un JAR exécutable. Il a besoin de plusieurs informations pour démarrer :

1. Pseudonyme à utiliser sur le chat ;
2. Emplacement du dossier de destination pour les transferts de fichiers ;
3. Addresse du serveur auquel se connecter ;
4. Port du serveur auquel se connecter.

Le client n'implémente que l'authentification « invité » , c'est-à-dire en donnant simplement un pseudonyme. Le protocole permet nonobstant de s'authentifier avec une combinaison pseudonyme et mot de passe. Le client fonctionne de la manière suivante, une console permet d'envoyer des messages. Elle permet également d'entrer des commandes comme par exemple `@<Pseudonyme>:<ID Serveur> <Message>` pour envoyer un message privé à un autre utilisateur, potentiellement sur un autre serveur ainsi que `/<Pseudonyme>:<ID Serveur> <Fichier>` pour envoyer un fichier à un autre utilisateur, potentiellement sur un autre serveur. Tout ce qui n'est pas reconnu comme une commande est considéré comme un message à envoyer publiquement.

3 Architecture

3.1 Paquets

Les paquets sont des records qui implémentent tous une interface `Packet`. Cette dernière offre deux méthode, une première qui retourne un `ByteBuffer` contenant les données du paquet, formattées selon sa structure, et une seconde qui accepte un visiteur.

En plus des deux méthodes à implémenter, il est bon d'implémenter une méthode statique qui retourne un `Reader` pour lire le paquet. Malheureusement, dû aux limitations de Java, il n'est pas possible de déclarer cette méthode dans l'interface comme abstraite pour forcer son implémentation.

3.2 Contexte abstrait

La gestion d'une connexion se fait par le biais d'un contexte. Un contexte permet de gérer autant l'aspect lecture que l'aspect écriture. En effet, les serveurs et clients fonctionnent en lecture/écriture simultanées. Afin de factoriser et d'éviter la duplication de code, un contexte abstrait implémente les fonctionnalitées basiques telles que la lecture, l'écriture des queues de paquets, ainsi que la mise à jour des intérêts (vonlonté de lire et/ou d'écrire).

Les clients et serveurs implémentent des contextes particuliers qui répondent à différents besoins en étendant du contexte abstrait.

3.3 Lecteurs et lecteurs « multi-part »

Afin de lire les octets efficacement, une interface `Reader` définis trois méthodes : une qui prend un `ByteBuffer` pour traiter les octets, une pour récupérer l'object lu lorsque la première méthode a retourner une valeur signifiant que le traitement est terminé, et une dernière pour réinitialiser le lecteur.

Quelques lecteurs simples permettent de simplement lire un entier, une chaîne de caractère, etc. À partir de ces lecteurs basiques, des lecteurs plus complexes peuvent voir le jour.

Enfin, pour lire des paquets entiers, il faut lire des parties simples telles que des entiers, des chaînes de caractères, etc. Afin d'éviter de dupliquer du code et de se retrouver avec une « boilerplate » conséquente, il existe un lecteur particulier qui prend une liste de lecteurs à déclencher, ainsi qu'une factory pour reconstruire le paquet. Ce lecteur s'appelle un `MultiPartReader`.

Les lecteurs pris en charge par le `MultiPartReader` sont des `PartReader`. Ces derniers prennent un lecteur à gérer ainsi qu'un consommateur qui prend l'objet lu. Chaque paquet implémente une instance anonyme d'un `AbstractPackerReader`. Ce dernier ne contient qu'une méthode qui retourne un `MultiPartReader`. Ceci permet de s'affranchir de la ré-instantiation du `MultiPartReader` à chaque solicitation.

Voici un exemple pour illustrer le fonctionnement. Soit le paquet `MsgPbl`, constitué d'un entier et de deux chaînes de caractères. L'implémentation anonyme du lecteur de paquet abstrait se présente de la façon suivante :

```
public static Reader<MsgPbl> getReader() {
    return new AbstractPacketReader<>() {

        // Champs permettant de stocker les différentes partie lues
        private int serverId;
        private String username;
        private String message;

        // Création du lecteur multi-part
        private final MultiPartReader<MsgPbl> reader = new MultiPartReader<>(List.of(
                // Factory retournant un PartReader<Integer>
                MultiPartReader.getInt(i -> serverId = i),
                // Factory retournant un PartReader<String>
                MultiPartReader.getString(s -> username = s),
                // Factory retournant un PartReader<String>
                MultiPartReader.getString(s -> message = s)
           // Factory permettant de reconstruire le paquet
        ), () -> new MsgPbl(serverId, username, message));

        @Override
        MultiPartReader<MsgPbl> reader() {
            return reader;
        }
    };
}
```

Le système est relativement complexe et hardu à expliquer. Voici donc un diagramme UML qui détaille les relations entre les lecteurs.

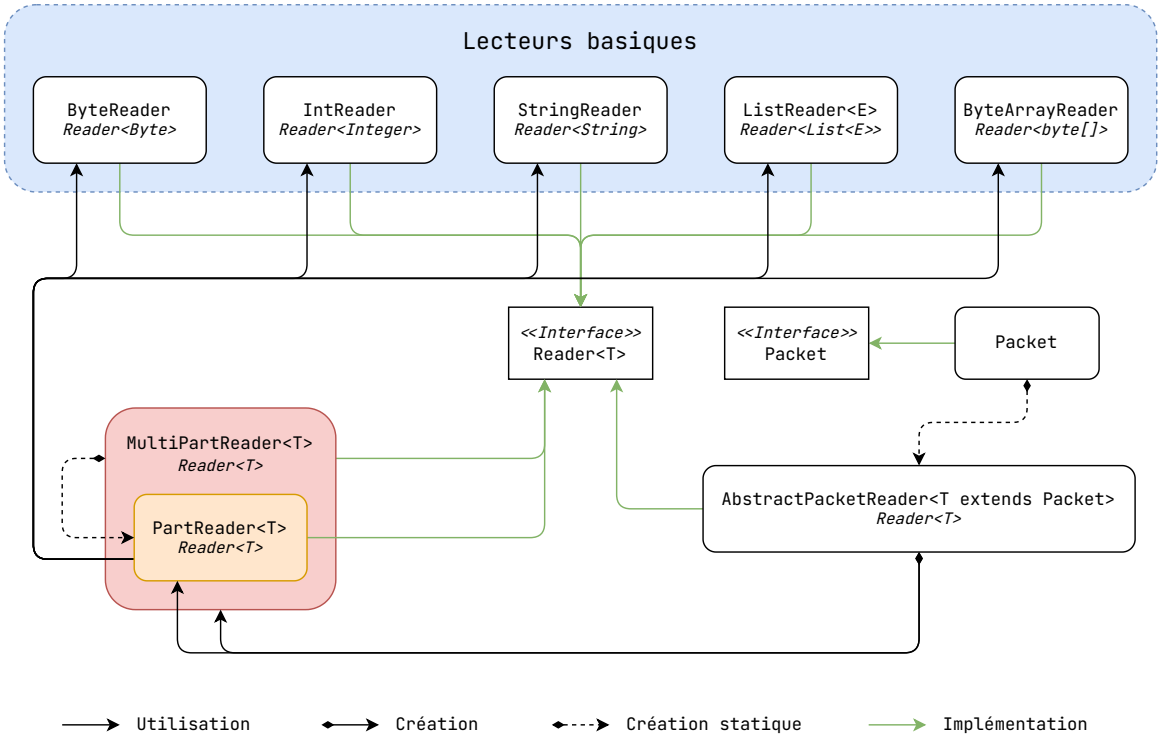

FIGURE 1 – Diagramme UML du système de lecteurs

Globalement, ce système est assez élaboré et complexe mais il permet d'obtenir des lecteurs complexes très simplement et rapidement. De plus, la lecture du code associé est simple et reflète immédiatement le comportement que le lecteur aura.

3.4 Gestion des paquets

Une fois les paquets lus en entrée, ils acceptent un `PacketVisitor`. Ce dernier permet d'implémenter des comportements à avoir pour chaque paquet. Le client ainsi que le serveur implémentent différemment le visiteur, et le serveur l'implémente plusieurs fois. De chaque côté, seul les paquets attendus sont gérés, les autres conservent leur comportement par défaut, c'est-à-dire la levée d'une `UnsupportedOperationException`.

Le client impléments un simple visiteur qui délègue certaines actions telles que l'authentification ou la réception de fichier au cœur du client, et pour le reste du temps affiche les données contenues dans le paquet, tel que pour les messages.

Le serveur a un fonctionnement plus complexe. En effet, lorsque le serveur accepte une connexion entrante, il peut s'agir d'un client ou d'un autre serveur. Lors de la création du contexte associé à la connexion, le serveur utilise un `DefaultVisitor`, qui implémente un comportement pour quelques paquets seulement tels que les paquets d'authentification et de fusion. Lorsqu'un tel paquet est reçu, on peut savoir s'il s'agit d'un client ou d'un serveur. Dès lors, le visiteur est remplacé par un visiteur plus explicite qui saura gérer les paquets propre au type de paire connectée.

4 Améliorations apportées

Depuis la soutenance bêta, la structure même du code a été revue du début. Ainsi les contextes ont été abstraits et factorisés, les paquets retravaillés en records associés à une interface commune, et le système de lecture complètement ré-imaginé pour permettre une implémentation la plus simple et concise possible, sans faire de compromis sur les performances.