Université Gustave Eiffel

Master Informatique

Rapport Technique

Programmation Réseau

Jimmy Teillard - Hadi Najjar

Groupe 2

Chargé de TD: Arnaud Carayol

M1

Mai 2022

Contents

1 Introduction

Ce rapport technique à pour but de compiler nos différents choix d'implémentation ainsi que l'architecture du projet. Il sera dédié une partie quant aux difficultés rencontrés, ainsi que de possible pistes d'auto-améliorations.

Ce projet a pour but d'appliquer les bases de la programmation réseau, ainsi que le design de protocol, et est réalisé dans le contexte d'un projet d'université.

ChatFusion est autant un protocole, qu'un client, et qu'un serveur. Le protocole en lui même et sa spécification sont définis dans le fichier RFC_CFP.txt à la racine du projet.

Pour faire simple, c'est un simple protocole de tchat textuel qui autorise les différents serveurs à se "fusionner" pour partager leur base d'utilisateurs connectés.

2 Compilation et Exécution

Pour compiler ce projet et l'exécuter, il est nécessaire d'avoir Java 17 (ou plus) installé sur votre machine. Les commandes explicitées ci-après supposeront que vous avez Gradle installé sur votre machine aussi ; cela dit, ce n'est pas une nécessité, et vous pouvez utilisé le wrapper script fourni à la racine du projet (gradlew et gradlew.bat).

Pour compiler, il suffit de lancer la commande suivante à la racine du projet :

```
gradle clean jar
```

Cela créera deux archives jar exécutables dans le dossier `executables/`.

Vous pouvez les lancer comme n'importe quelle autre archive jar :

2.1 Serveur

`java -jar ChatFusion-server-[version].jar [PORT] [SERVERNAME]`

Le serveur a besoin d'un port sur lequel il écoutera les connections, ainsi qu'un nom unique composé de 5 caractères ASCII non nuls qui servira à l'identifier.

Une fois un serveur lancé, plusieurs commandes sont utilisables :

- `INFO` : affiche les infos du serveur (notamment son leader, son nombre de clients, et son nombre de frères.
- `TOGGLE DEBUG` : active/desactive l'affichage de message de debug.
- `SHUTDOWN` : stoppe l'acceptation de nouvelles connexions entrantes
- `SHUTDOWN NOW` : stoppe le serveur immédiatement de manière propre.
- `FUSION [hostname] [port]` : enclenche une demande fusion avec le serveur donné

2.2 Client

`java -jar ChatFusion-client-[version].jar [HOSTNAME] [PORT] [FILES_DIR] [USERNAME]`

Le client a besoin d'une adresse et d'un port (de serveur) sur lequel se connecter, ainsi qu'un dossier dans lequel les fichiers téléchargés iront, et d'un nom d'utilisateur unique composé de caractères ASCII non nuls qui servira à l'identifier. Si un client se connecte sur un serveur avec un nom d'utilisateurs déjà utilisé, la connexion sera refusée.

Une fois un client lancé et connecté, plusieurs commandes sont utilisables :

- `[message]` : Envoie un message publique (reçu par tout le monde)
- `@[username]:[servername] [message]` : envoie un message privé à l'utilisateur d'un serveur donné.
- `/[username]:[servername] [message]` : envoie un fichier (privé) à l'utilisateur d'un serveur donné.
- `:quit` : ferme l'application de manière propre.

3 Architecture

3.1 Sections

Le projet est un multi-module découpé en trois sections :

- core : qui contient les fichiers communs aux deux autres sections, notamment les interfaces de Context et FrameVisitor, ainsi que le modèle des différents types de paquets et leur readers.

- client : qui contient les fichiers spécifiques au client.
- server : qui contient les fichiers spécifiques au serveur.

core est une dépendance des deux autres sections (au sens de Gradle).

3.2 Core

3.2.1 Modèle

Le modèle représente les différents paquets qui transitent sur le réseau. On les appellera des frames.

Une frame n'est qu'un POJO composé d'un autre POJO élémentaire qui représente le contenu d'un paquet structurellement. On les appelle des parts. Une part est capable to donner sa représentation sous forme de ByteBuffer.

3.2.2 Reader API

Ce projet contient une API complète et modulaire pour créer des reader de ByteBuffer. En effet, un Reader est un objet qui est capable de traiter un buffer en plusieurs fois, et de donner son état. Quand un reader a terminé, il peut retourner un objet du type de son type paramétré.

L'implémentation centrale est le FrameReader, qui, comme son nom l'indique, est capable de traiter des buffers qui représentent n'importe quel type de frame. Pour cela, il possède des instances de PartReader, dont le rôle est de traiter de buffers représentant des parts (par exemple un IdentifiedMessage, qui peut être contenu dans PrivMsg ou dans un PrivMsgResp), auxquels il délègue le traitement du buffer donné en entrée, en fonction de l'opcode qui précède le reste du buffer. (Chaque opcode représente un type de frame, et sont tous spécifiés dans la RFC ainsi que dans fichier OpCodes).

Pour faciliter la création de nouveaux types de readers, plusieurs classes sont disponibles et rassemble les cas principaux de readers.

- PrimitiveReader (abstract) : ces readers représentent les readers "atomiques", dans le sens qu'ils ne sont pas une composition d'autres readers ; ils connaissent exactement le nombre de bytes à lire pour être complétés.
- SizedReader (abstract) : ces readers représentent les readers qui doivent lire un objet d'une taille variable, typiquement une String. Ils lisent d'abord un entier qui représente une taille, puis ce nombre de bytes pour construire l'objet voulu.
- ListReader (abstract) : ces readers sont très similaires aux SizedReaders, mais produisent une liste d'objet d'une taille variable.
- PartReader (abstract) : ces readers lisent des parts. Ils n'ont rien de réellement spécifique et servent de wrapper de données (des builders en quelque sorte) ainsi que provider d'un type de reader qui lui même traite une part. Son intérêt est lié au type de reader suivant.

- ComposedReader (factory) : à la fois un reader et une factory class, ce reader complexe permet de créer très simplement des readers en composant d'autres readers de n'importe quel type. En effet, plusieurs readers se servent simplement d'autres readers successivement de facon très similaire. Ainsi, un composed reader n'a besoin que d'une série de reader ainsi que de fonctions anonymes de construction (des consumer, puis un producer final). Il est nécessaire que cette série de readers soient wrapped à l'intérieur de InnerReader, pour y associer les fonctions de construction (Une Map aurait pu être utilisée, mais on a préféré un objet spécifique pour des raisons sémantiques et de consistance). L'intérêt des PartReaders se dévoile donc ici, quant aux fonctions de construction : les lambdas java ne permettent pas la mutation de référence de variables intra-function-scope, pour cela, il est soit nécessaire de faire des mutations d'objets (et non de référence), ce qui est très peu pratique dans le cas des types primitifs et de String (qui sont tous non mutables). PartReader devient alors cet objet à muter par ses champs.

Il existe aussi un InetAddressReader qui n'est issu d'aucun des readers ci-dessus, car il ne s'applique à aucun pattern répétable (à cause d'un embranchement après lecture de la version d'IP).

3.2.3 Context API

Que cela soit pour le serveur ou le client, une connexion, peu importe son type (client-serveur, serveur-client, ou serveur-serveur), il est nécessaire d'englober une logique commune : on doit pouvoir lire sur cette connexion, écrire, ou bien la fermer. C'est ce que représente l'interface Context.

A cet effet a été fournie une classe abstraite AbstractContext qui rassemble le comportement commun de n'importe quel contexte. En effet, dans tout les cas, la réception et l'envoi de bytes sur le réseau sont toujours les mêmes. La seul chose qui diffère est alors ce que l'on fait des paquets (frames) lues. De cette façon, chaque instance de Context n'a généralement besoin que de savoir queue des frames à envoyer (sous forme de ByteBuffer), ainsi qu'une façon spécifique de traiter les frames reçues. Cette idée est résolue par l'API présentée dans la sous section suivante.

3.2.4 FrameVisitor API

Comme dit précédemment, il est nécessaire que, suivant le type de connexion, les frames reçues soient traitées différemment. Pour cela, chaque frame est aussi un visitable. C'est à dire qu'il existe une interface FrameVisitor qui contient une méthode "visit" pour chaque frame qui se lance respectivement pour cette dernière lorsque l'on demande a ce qu'une frame accepte ledit visiteur. En particulier, le comportement par défaut d'un FrameVisitor, pour chaque frame reçue, est de fermer la connexion du contexte auquel il est lié. Ainsi, lorsqu'un contexte reçois un paquet inattendu, la connexion sera terminée.

3.3 Client

L'implémentation du client en ce qui concerne le contexte (client-serveur) et le visiteur n'a rien de notable en particulier, si ce n'est la gestion de l'envoi de fichiers intercalé entre les envois de messages publiques et privés.

En effet, deux classes se chargent de ce méchanisme : FileSender et FileManager. Le rôle de la première est d'ouvrir un fichier donné, et de le découper en chunk pour les ajouter à une queue spécifique au fichiers dans le contexte concerné. En effet, le contexte va alors pull alternativement les paquets de la queue normale, et de la queue de fichiers. Quant au FileManager, son rôle sera d'écrire dans un fichier temporaire à chaque fois que le client recevra un chunk, jusqu'à ce que le fichier soit complet (il sera alors validé et renommé à ce moment).

3.4 Serveur

3.4.1 Contexte et Visiteurs

La particularité du serveur est que l'origine contextuelle d'une connexion entrante est inconnue tant que l'on a pas reçu de paquet ; en effet, un serveur ne peut recevoir un paquet TEMP (login temporaire) que d'un client, alors qu'il ne peut recevoir un FUSIONREQ (demande de fusion) que d'un autre serveur. Afin de gérer ce problème, nous avons décidé de faire en sorte que le contexte reste générique (il contient toujours un type enuméré UNKNOWN/CLIENT/SERVER à des fins d'assertions), mais possède un visiteur variable. En effet, on aura alors trois types de visiteurs : Un visiteur qui représente une connexion de type encore inconnu, un autre qui représente une connexion serveur-client, et enfin un dernier qui représente une connexion serveur-serveur. L'intérêt de cette approche est que l'on ne fait varier que le comportement du contexte, et non pas ses données (en l'occurrence, ses queues). Cela permet de garantir un état cohérent pour chaque contexte ; aussi, il est à noter que cela permet, pour un client par exemple, de lui interdire d'envoyer des messages tant qu'il n'a pas envoyé de paquet TEMP (pour dire qu'il est bien un client), car le comportement par défaut est de fermer la connexion lorsqu'un paquet non reconnu par le visiteur est reçu.

Il est aussi à noter qu'un client n'a alors pas besoin de s'identifier à chaque envoi de paquet, puisque son identité est directement figée et liée à son visiteur et contexte : il est impossible pour un client de se faire passer pour quelqu'un d'autre.

3.4.2 Fusion

L'autre axe majeur du serveur est la gestion de la fusion. Pour cela, nous avons fait le choix de représenter une fusion de serveurs par un maillage fort. C'est à dire qu'un réseau de serveurs forme un graphe non dirigé complet en terme de connexions (en d'autres termes, chaque serveur est directement connecté à tout les autres).

Ce choix à d'abord été fait pour simplifier grandement la diffusion inter-serveurs de message publiques, privés, et de fichiers : un paquet ne fait au plus qu'un seul saut avant d'arriver à destination.

Cependant, cela a créé des difficultés quant à la procédure de fusion elle-même. Il a été nécessaire de la faire en plusieurs étapes : L'initialisation de la demande transférée au leader d'un groupe (choisi par son nom, en premier dans l'ordre alphanumérique), l'ouverture de la connexion vers le serveur leader opposé, la phase de verrou (pour empêcher d'autres serveurs de demander à fusionner lorsqu'une fusion se déroule), la fusion en elle même où chaque serveur tente d'initier une connexion à tout les autres (et qui doit être validée), et enfin la phase de déverrouillage quand tout est terminé. C'est un processus très lourd et très prône à confusion.

4 Choix, Difficultés, et Améliorations

Une des améliorations apportées, déjà mentionnée précédemment dans ce document, à été la gestion de la réception d'un message par un serveur, qu'il doit transmettre aux clients (et serveurs) concernés. On avait pour cela plusieurs overloads de méthodes pour chaque type de paquet. Dorénavant, il ne reste plus que deux méthodes : broadcast et sendTo, qui respectivement transmet le message à tout les clients (et serveur si le boolean de broadcast est à true), et transfère un message à un utilisateur désigné.

```java
@Override
public void visit(Msg frame) {
    var msgBuffer :ByteBuffer = new IdentifiedMessage(
        new Identifier(username, server.name()),
        frame.message()
    ).toBuffer();
    server.broadcast(msgBuffer, forward: true);
}

@Override
public void visit(PrivMsg frame) {
    var message :IdentifiedMessage = frame.message();
    var msgBuffer :ByteBuffer = message
        .with(new Identifier(username, server.name()))
        .toBuffer();
    server.sendTo(
        message.identifier(),
        msgBuffer,
        OpCodes.PRIVMSGRESP,
        OpCodes.PRIVMSGFWD
    );
}

@Override
public void visit(PrivFile frame) {
    var fileChunk :IdentifiedFileChunk = frame.identifiedFileChunk();
    var fileBuffer :ByteBuffer = fileChunk
        .with(new Identifier(username, server.name()))
        .toBuffer();
    server.sendTo(
        fileChunk.identifier(),
        fileBuffer,
        OpCodes.PRIVFILERESP,
        OpCodes.PRIVFILEFWD
    );
}
```

Il est à noter qu'il n'est pas nécessaire de valider la provenance d'un message puisque l'identité d'un utilisateur est fortement liée à la connexion représentée par un contexte. Si la connexion se ferme, alors son identification aussi. Par conséquent il est impossible qu'un utilisateur se fasse passer pour un autre une fois connecté. D'autre part, au vu de l'utilisation de deux visiteurs différents pour représenter une connexion non identifiée et un utilisateur connecté, il est impossible pour un client de commencer à envoyer des messages alors qu'il n'est pas connecté.

Un des choix majeurs que nous avons fait a été de représenter une fusion par un maillage fort, à regret. En effet, le bénéfice post-fusion est incomparablement faible face à la difficulté de la mise en place de ladite fusion. Cela a demandé une rigueur conséquente qui aurait pu être évitée.

Cela dit, c'en est tout de même resté une amélioration comparé à la beta, puisque l'organisation des comportements axés sur les visiteurs plutôt que sur plusieurs types de contextes à permit de grandement simplifier les différentes méthodes du serveur. Par exemple, la diffusion de message (peut importe son type) est toujours gérée de la même manière, dans une seule fonction, au lieu de plusieurs dans les cas précédents. En revanche, il reste un cas de fusion non fonctionnel : lorsqu'un leader seul demande à fusionner avec un non leader d'un groupe de taille supérieure à 3 (oui, c'est très spécifique, et c'est sa spécificité qui fait que nous n'avons pas réussi à le résoudre) ; cela dit, faire fusionner deux groupes de 2 ou plus, en initiant de non leader à non leader, fonctionne parfaitement bien, donc le problème ne vient pas tant du nombre (?).

Un autre axe d'amélioration consistait à simplifier le transfert de fichiers coté client. Auparavant, nous créions un thread par fichier à envoyer, qui s'occupait de queue les chunks du fichier périodiquement (possiblement intercalé de messages classiques). Cependant, c'est un problème dans le sens que cette solution amène à des questions de performance et de sécurité. Si quelqu'un décide d'envoyer beaucoup de fichiers en même temps, il risque d'ouvrir énormément de threads en même temps, ce qui n'est absolument pas désirable. Dorénavant, il n'y a plus de thread différent pour l'enqueuing de chunk. A la place, les contextes possèdent dorénavant une queue secondaire qui doit être pulled alternativement de la queue principale lors de l'envoi de paquets. Cela permet aussi de faire en sorte que l'on ai un système de priorité de queues.

```java
private void processQueue(ArrayDeque<ByteBuffer> queue, Runnable onFinish) {
    var buffer :ByteBuffer = queue.peek();
    assert buffer != null;
    if (!buffer.hasRemaining()) {
        queue.poll();
        return;
    }
    Readers.read(buffer, bout);
    onFinish.run();
}

protected void processOut() {
    while ((!queue.isEmpty() || !filesQueue.isEmpty()) && bout.hasRemaining()) {
        if (queue.isEmpty()) processQueue(filesQueue, () -> sentMessages = 0);
        else if (filesQueue.isEmpty() || sentMessages < CONSECUTIVE_MSG_LIMIT) processQueue(queue, () -> sentMessages++);
        else processQueue(filesQueue, () -> sentMessages = 0);
    }
}
```

Encore lié au fichiers, mais cette fois-ci à la réception, auparavant, nous gardions en mémoire des listes de tout les chunks reçus tant que l'on ne les avais pas tous reçus pour un fichier donner. Encore une fois, cela pose des problèmes de performance. Après réflexion, nous avons décidé de nous inspirer des navigateurs tels que Google Chrome ou Firefox dans la façon dont ils téléchargent des fichiers : en effet, lorsqu'un fichier est en cours de téléchargement, un fichier temporaire (souvent terminé par .tmp ou .part) est créé et est rempli au fur et à mesure que les chunks sont recus. Une fois le fichier complété, l'extension .tmp/.part est retirée du nom pour signifier à l'utilisateur que son fichier a bien été reçu. C'est exactement le processus du FileManager. Pour résumer, au lieu de stoquer potentiellement plusieurs listes de plusieurs chunks, on ne stock que les FileOutputStream des fichiers encore non complétés.

Finalement, la dernière amélioration apportée au projet depuis la beta a été la création d'une API complète pour les readers. Au lieu de répéter le même code constamment, il est maintenant possible d'utiliser des patterns de reader très facilement. Les détails de la nouvelle implémentation ont été expliqués dans une section précédente. Voici un exemple d'utilisation d'un ComposedReader au sein d'un PartReader :

```java
public class ServerInfoReader extends PartReader<ServerInfo> implements Reader<ServerInfo> {
    private String servername;
    private InetAddress ip;
    private short port;

    @Override
    protected Reader<ServerInfo> provide() {
        return ComposedReader.with(
            () -> new ServerInfo(servername, ip, port),
            inner(new ServernameReader(), v -> servername = v),
            inner(new InetAddressReader(), v -> ip = v),
            inner(new ShortReader(), v -> port = v)
        );
    }
}
```

5 Conclusion

Ce projet nous a permit d'observer beaucoup d'aspects du développement d'application, ainsi que la gestion de paradigmes non bloquants, en plus d'appliquer plusieurs design patterns. En effet, la première phase de conception du protocole était très enrichissante d'un point de vue gestion de projet, en particulier de cahier des charges. Il nous a aussi permit de voir que certaines décisions peuvent être regrettées (en l'occurrence, en ce qui concerne la fusion).