HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Commusica

Le lecteur de musique communautaire et égalitaire

Rapport final

Groupe 4B

Chef de projet	Ludovic Delafontaine
Chef remplaçant	Lucas Elisei
Membres	David Truan
	Denise Gemesio
	Thibaut Togue
	Yosra Harbaoui
Professeur	René Rentsch

HEIG-VD - Semestre d'été 2017

Table des matières

Table des figures

1 Introduction

Durant le quatrième semestre de la section TIC de l'HEIG-VD, nous devons effectuer un projet par groupes de cinq ou six personnes. Le but est de mettre en oeuvre les connaissances que nous avons acquises au long des semestres précédents à travers un projet conséquent. Nous devrons prendre conscience des difficultés liées au travail de groupe, ainsi qu'apprendre à planifier un travail sur plusieurs mois. Au terme du semestre, nous devons rendre un programme complet et fonctionnel, avec une documentation adéquate et être capables de le présenter et le défendre.

Le projet dure seize semaines et vaut trois crédits. Un crédit valant 30 heures de travail, le temps de travail est de 540 heures pour toute l'équipe, soit cinq heures et demi par membres du projet, par semaine.

Dans le cadre du projet, l'équipe de programmation est composée du chef de projet Ludovic Delafontaine, de son remplaçant Lucas Elisei et des membres David Truan, Thibaut Togue, Yosra Harbaoui et Denise Gemesio.

Dans ce rapport, nous allons expliquer notre démarche de travail et les principaux choix d'architecture et de design de code. Il sera structuré selon les principaux paquets de notre application.

Ce rapport, étant à rendre deux semaines avant la fin du temps total alloué pour le projet, s'arrête donc en semaine quatorze. Les deux dernières semaines seront consacrées à la défense orale, étape qui ne sera pas expliquée dans ce document.

2 Objectif

Le but de notre programme est de proposer une application client-serveur qui permettra aux clients d'envoyer des fichiers musicaux au serveur pour que celui-ci les joue. Il se démarque d'une simple application de lecture en continu (streaming) dans le fait que la liste de lecture ne peut être changée que par les clients, par le biais d'un système de votes positifs ou négatifs. Ceux-ci permettent à un morceau d'être placé plus en avant ou en arrière dans la liste de lecture. Cela permet donc à chacun de donner son avis, tout en centralisant la lecture de la musique sur un seul ordinateur. De plus, l'application met à disposition les fonctionnalités suivantes pour une expérience encore plus communautaire :

- Passer au morceau suivant, si la majorité le souhaite. Cela aura pour effet de récupérer le morceau de musique suivant la plus demandée et de la lire.
- Augmenter et diminuer le volume, si la majorité le souhaite.
- Système de favoris afin de permettre aux utilisateurs de sauvegarder les informations d'un morceau de musique pour pouvoir le retrouver par la suite.

Nous souhaitons répondre à l'éternel problème qu'est le fait de devoir se partager une prise jack ou de devoir se battre pour pouvoir passer un morceau que l'on souhaite écouter.

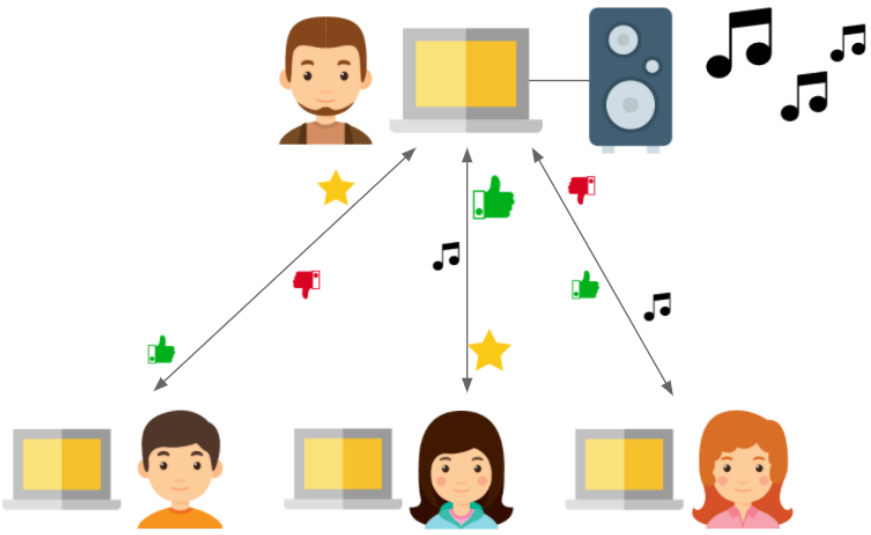

Fig. 1: Fonctionnement général du programme

3 Public cible

Cette application visera un public dont la gestion de la musique lors d'un événement est critique. Cela peut autant concerner les lieux publics, tels que les bars, ou encore, les événements privés, tels que les soirées entre amis.

Notre application se voudra donc simple à utiliser, afin qu'un néophyte dans le domaine de l'informatique puisse l'utiliser.

4 Conception et normes

Avant de nous lancer dans le développement, nous avons pris le temps de réfléchir à l'architecture de notre programme. Nous avons souhaité séparer au mieux les différentes entités de notre application, afin de simplifier le développement, la compréhension du programme et rester très abstraits.

Avant de commencer le développement, nous avons donc passé trois semaines à définir l'architecture du programme décrite au chapitre suivant.

Une fois l'architecture bien définie, nous avons déterminé des normes de développement afin d'utiliser toutes et tous la même façon de coder. Parmi les points discutés, nous avons abouti aux conventions suivantes :

- Langue de documentation : la langue véhiculaire de l'informatique étant l'anglais, il était évident que le développement allait se faire dans cette langue. C'est la raison pour laquelle notre code, ainsi que la JavaDoc, sont écrits en anglais.
- Utilisation de la JavaDoc pour la génération future d'une documentation technique.
- Définition de la syntaxe à utiliser pour les variables, constantes et autres structures syntaxiques.

5 Architecture

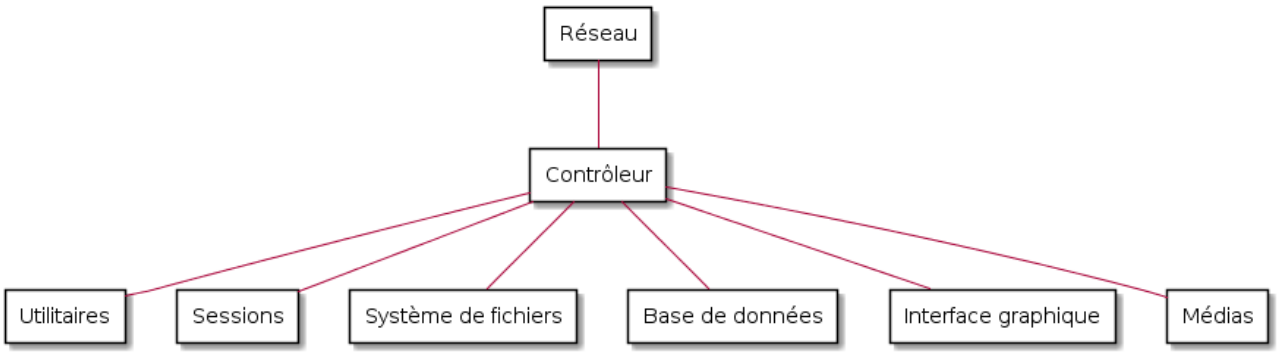

Fig. 2: Architecture du programme

5.1 Entité *Réseau*

Cette entité permet la gestion du réseau entre les clients et le serveur. Elle permet de répondre aux attentes suivantes :

- Le serveur doit pouvoir gérer plusieurs clients, mais sans devoir garder une connexion constante entre chaque client et le serveur.
- Un serveur et un client doivent pouvoir communiquer l'un avec l'autre en utilisant une liaison de communication "privée" à l'aide de l'Unicast.
- Un serveur doit pouvoir diffuser à tous les clients un message, afin que tout le monde le réceptionne et le traite à l'aide du Multicast.

5.2 Entité *Contrôleur*

Cette entité permet le contrôle de l'application. C'est elle qui va gérer son comportement, faire la liaison entre toutes les entités et répondre aux demandes des utilisateurs et des serveurs.

5.3 Entités *Utilitaires*

Ces entités sont celles qui ne trouvent pas leur place dans des entités spécifiques. L'entité de configuration, qui permet de récupérer des propriétés d'un fichier de configuration, fait notamment partie de ces entités utilitaires.

5.4 Entité *Sessions*

Cette entité permet de gérer les notions de session qui servent à connaître les personnes connectées ou les serveurs accessibles. Il y a notamment deux notions de session : les sessions utilisateur et les sessions serveur.

Dans le cas des sessions utilisateur, le but est de pouvoir limiter un utilisateur dans son nombre d'actions sur le serveur - voter pour un morceau, voter contre un morceau, faire une demande de changement de musique, etc. - et savoir combien d'utilisateurs sont actifs sur le serveur afin de savoir quand une action définie par une majorité doit avoir lieu.

Dans le cas des sessions serveur, le but est de savoir si un serveur est encore accessible. A chaque mise à jour de la part du serveur, la session associée sera mise à jour. Si l'un des serveurs venait à être éteint ou déconnecté, le client supprimerait le serveur afin qu'il ne tente pas d'y accéder.

5.5 Entité *Système de fichiers*

Cette entité a pour rôle d'assurer la gestion des fichiers, en interagissant avec le système de fichiers. Elle permet de sauvegarder, supprimer, renommer des fichiers sur le disque, recevoir des fichiers par réseau et vérifier qu'ils ne soient pas corrompus.

5.6 Entité *Base de données*

Cette entité est une abstraction de la base de données. Elle permet de simplifier l'interaction avec cette dernière en mettant à disposition des méthodes pour les opérations de base sur la base de données.

Cette entité permet à un utilisateur, par exemple, de sauvegarder les métadonnées des morceaux qui lui plaisent dans la base de données.

Les différentes tables sont générées automatiquement par Hibernate (voir `Technlogies utilisées` et la corrélation entre les classes Java et la base de données est définie dans les fichiers `.hbm.xml`, dans le dossier `ressources/models`), ce qui permet de séparer les deux notions : programmation orientée objet et base de données.

5.7 Entité *Interface graphique*

Cette entité représente ce que nous allons montrer à l'utilisateur, afin qu'il ait une interface pour intéragir avec le programme. Cette interface sera liée directement au contrôleur, qui saura quoi faire en fonction de l'action demandée.

5.8 Entité *Médias*

Cette entité regroupe tout ce qui concerne les médias de notre application : morceaux de musique, listes de lecture, lecteur de musique.

6 Description technique

La description technique se développera dans l'ordre croissant de complexité des différentes entités de notre programme.

Pour chacun des paquets, dérivés des entités décrites au chapitre précédent, nous décrirons brièvement le but des différentes classes.

6.1 Paquet `database`

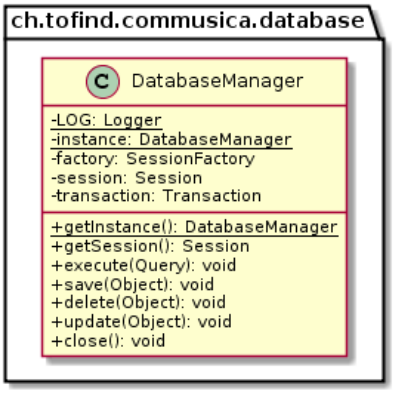

Fig. 3: Paquet `database`

Ce paquet est constitué essentiellement de la classe `DatabaseManager`, dont le rôle est d'assurer les méthodes définies par les notions CRUD - Create, Read, Update, Delete - de la base de données de notre application et d'assurer la fermeture de la connexion à celle-ci.

Hibernate est la librairie utilisée dans notre projet afin de pouvoir communiquer avec la base de données. Elle utilise un système de sessions afin de garder en cache les différentes informations de la base de données. Avant d'enregistrer quoi que ce soit dans cette dernière, Hibernate va interroger son cache afin de savoir si l'information est disponible et s'il est nécessaire de faire une quelconque action sur la base de données réelle.

Fig. 4: Schéma de la base de données

6.2 Paquet `file`

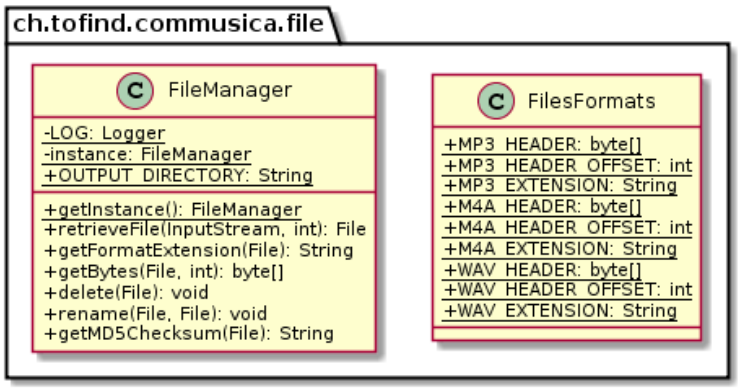

FIG. 5: Paquet `file`

Le paquet `file` est constitué de deux classes : `FileManager` et `FilesFormats`.

6.2.1 `FilesFormats`

Actuellement, notre application supporte trois formats de fichiers audio : MP3, M4A et WAV. Cette classe permet de définir les caractéristiques d'un fichier, c'est-à-dire, tous les éléments nous permettant de connaître le type de fichier.

6.2.2 `FileManager`

Cette classe permet de supprimer, stocker et déterminer le type de fichier. Pour retrouver l'extension du fichier, nous avons procédé de la manière suivante :

- Pour les fichiers MP3, nous regardons les trois premiers octets, depuis le début du fichier.
- Pour les fichier M4A, nous regardons les premiers octets, en partant du quatrième octet, depuis le début du fichier.
- Pour les fichiers WAV, nous regardons à partir du huitième octet, depuis le début du fichier.

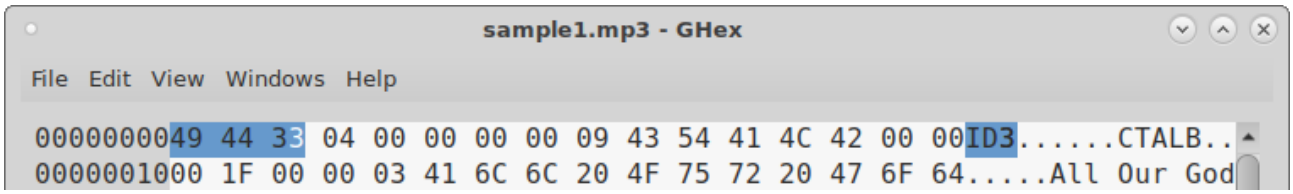

FIG. 6: Aperçu hexadécimal d'un fichier MP3

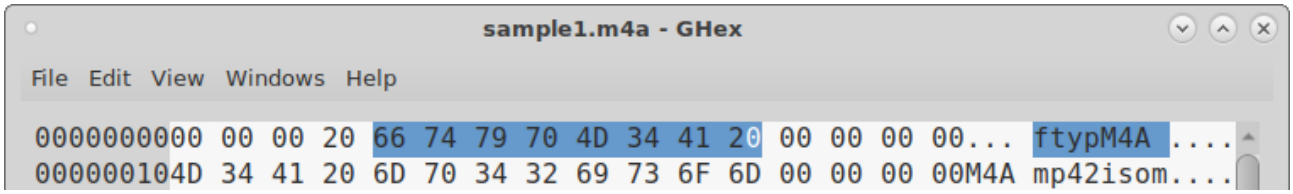

FIG. 7: Aperçu hexadécimal d'un fichier M4A

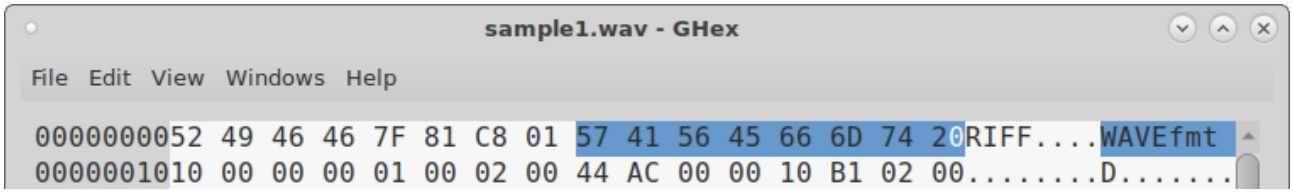

FIG. 8: Aperçu hexadécimal d'un fichier WAV

Connaître le type de fichier nous permettra de traiter uniquement les fichiers supportés par notre plateforme et, aussi, en termes de sécurité, éviter qu'un utilisateur ne fasse planter le serveur en envoyant un fichier qui n'est pas supporté par celui-ci.

6.3 Paquet `network`

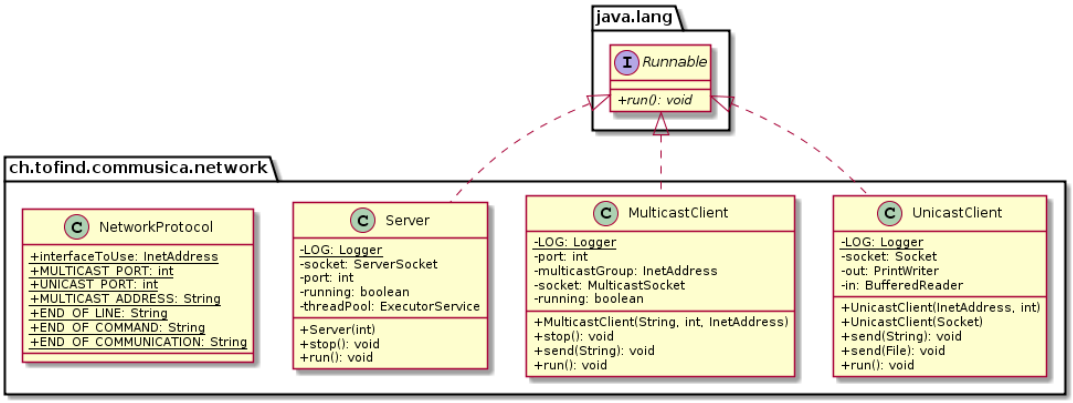

FIG. 9: Paquet `network`

Pour répondre aux besoin architecturaux vus au chapitre 5, nous avons développé plusieurs classes qui s'occupent de gérer les sockets et envois de commandes via le réseau. Nous savions que notre protocole enverrait du texte, car la lecture de ligne dans un flux d'entrée se fait facilement grâce à la méthode `readline()`. Nous avions également besoin d'envoyer des objet sérialisés au format JSON.

6.3.1 Protocole et commandes

La plupart des commandes listées ci-dessous sont disponibles dans la classe `ApplicationProtocol`, mais, comme elles sont envoyées par le réseau, nous avons préféré les expliquer dans ce chapitre. Toutes les commandes envoyées par Unicast ont comme deux premiers arguments :

- L'ID de l'expéditeur, qui est le hash de l'adresse MAC de l'interface réseau qu'il utilise.
- Le socket utilisé pour la communication, qui est rajouté à la réception, avant d'envoyer la commande au `Core` (la notion de `Core` est développée au chapitre `Core`).

De ce fait, les tableaux ci-dessous n'indiquent, dans leur colonne `Arguments`, que les arguments en plus de ces deux dans l'ordre dans lequel ils sont envoyés.

La seule commande envoyée en Multicast est `PLAYLIST_UPDATE`, qui est envoyée depuis le serveur à tous les client.

Les objets décrits ci-dessous sont évidemment sérialisés avant d'être transférés sur le réseau.

Commande	Arguments	But
NEW_ACTIVE_CLIENT	Aucun	Envoyée lors de sa première connexion à un serveur.
TRACK_REQUEST	Track à envoyer	Permet de savoir s'il faut envoyer le fichier ou si le système l'a déjà.
SENDING_TRACK	Taille du fichier à envoyer Track à envoyer	Envoi du fichier en réponse à `TRACK_ACCEPTED` si accepté.
PLAY_PAUSE_REQUEST	Aucun	Souhait de mettre le morceau actuel en lecture/pause.
NEXT_TRACK_REQUEST	Aucun	Souhait de passer au morceau suivant.
TURN_VOLUME_UP_REQUEST	Aucun	Souhait d'augmenter le volume.
TURN_VOLUME_DOWN_REQUEST	Aucun	Souhait de baisser le volume.
UPVOTE_TRACK_REQUEST	ID de la Track à upvoter	Souhait d'upvoter un morceau de la liste de lecture.
DOWNVOTE_TRACK_REQUEST	ID de la Track à downvoter	Souhait de downvoter un morceau de la liste de lecture.
END_OF_COMMUNICATION	Aucun	La communication doit être stoppée.

Fig. 10: Commandes envoyées par le client au serveur

Commande	Arguments	But
PLAYLIST_UPDATE	L'adresse du serveur Le nom du serveur	Notifier tous les clients en Multicast de l'état du système.
TRACK_ACCEPTED	Aucun	Réponse à la commande `TRACK_REQUEST` lorsque le morceau est accepté.
TRACK_REFUSED	Aucun	Réponse à la commande `TRACK_REQUEST` lorsque le morceau est refusé.
TRACK_SAVED	Aucun	Réponse à `SENDING_TRACK` si le morceau a été enregistré avec succès.
TRACK_UPVOTED	Aucun	Réponse à la commande `UPVOTE_TRACK_REQUEST` si le morceau a été upvoté.
TRACK_DOWNVOTED	Aucun	Réponse à la commande `DOWNVOTE_TRACK_REQUEST` si le morceau a été downvoté.
SUCCESS	Message de succès	Envoie un message de succès au client lors du succès d'une commande
ERROR	Message d'erreur	Envoie un message d'erreur au client lors de l'erreur d'une commande.
END_OF_COMMUNICATION	Aucun	Commande indiquant que la communication doit être stoppée.

Fig. 11: Commandes envoyées par le serveur au client

6.3.2 `Server`

Côté serveur, nous avons décidé d'opter pour une architecture avec un thread réceptionniste `Server` qui va attendre une nouvelle connexion de la part des clients. Une fois un client arrivé, il va lancer un thread `UnicastClient` qui va s'occuper de la communication avec le client. Cette communication se fait via un socket Unicast, car il s'agit d'une communication privée entre le serveur et le client. Nous avons choisi cette solution, car plusieurs connexions avec des clients peuvent survenir simultanément et ce système réceptionniste, avec un thread par client, gère plusieurs connexions en même temps, contrairement à un système avec un seul thread qui s'occupe d'un client à la fois.

6.3.3 `UnicastClient`

La classe `UnicastClient` va pouvoir recevoir les commandes venant du réseau et en renvoyer. Elle implémente l'interface `Runnable`, ce qui lui permet de s'exécuter en tant que thread. Sa force réside dans le fait qu'elle peut être utilisée aussi bien du côté serveur que du côté client, grâce au fait qu'elle lit les commandes reçues et les envoie au `Core` pour qu'il les exécute.

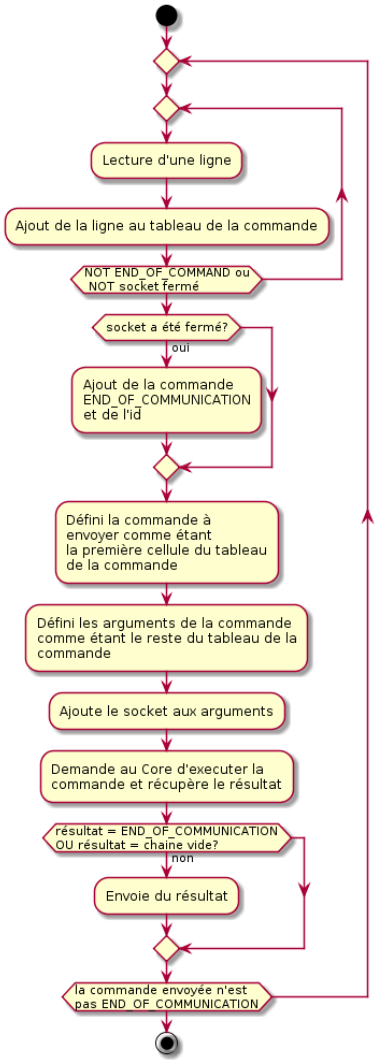

Fig. 12: Diagramme d'activité de la réception et de l'envoi Unicast

Ce diagramme montre la lecture d'une commande venant du réseau et son découpage pour en extraire les arguments et la passer au `Core`. Celui-ci s'occupera d'exécuter la commande si elle est disponible dans l'instance de son `AbstractCore` (voir chapitre `Core`). Le `Core` renvoie ensuite une commande à envoyer en réponse à celle reçue. La communication se termine lorsque l'une des extrémités envoie la commande `END_OF_COMMUNICATION` ou qu'elle ferme son socket.

6.3.4 `MulticastClient`

Cette classe implémente aussi l'interface `Runnable` dans le but de lancer son exécution dans un nouveau thread. Comme pour `UnicastClient`, cette classe peut être utilisée du côté serveur comme du côté client. La méthode `run()` permet au thread de rejoindre le groupe Multicast à l'adresse définie dans le `NetworkProtocol`. Le thread va ensuite attendre de recevoir des datagrammes venant du groupe Multicast jusqu'à son arrêt par la méthode `stop()`.

Nous avons été confrontés à un problème lors du développement, lorsque nous nous sommes rendus compte qu'un `MulticastSocket` utilisait la première interface réseau disponible sur l'ordinateur plutôt que celle qui était réellement connectée. Cela prenant énormément de temps à être résolu, nous avons décidé de mettre à disposition un choix d'interfaces réseau dans les interfaces graphiques de l'utilisateur et du serveur.

Les `Core` ont chacun un `MulticastClient` pour pouvoir envoyer la liste de lecture actuelle ainsi que les informations sur l'état du lecteur - en pause ou en lecture, état du volume, etc.

Pour utiliser le Multicast, il est nécessaire de spécifier quelle est l'interface réseau à utiliser pour l'émission et la réception des données. Dans le cas où la machine a plusieurs interfaces réseau, il faudra faire attention et sélectionner la bonne interface dans la liste de l'interface graphique.

6.3.5 `NetworkProtocol`

C'est dans cette classe que sont définies les commandes spécifiques au réseau : `END_OF_COMMUNICATION` et `END_OF_COMMAND`. Les ports pour les différents sockets ainsi que l'adresse du groupe Multicast se trouvent également dans cette classe. Ces derniers ont été choisis arbitrairement parmi les plages disponibles. Bien que peu probable, il est possible qu'une autre application utilise ces mêmes ports. Dans ce cas, le bon fonctionnement de Commusica se retrouverait corrompu. Nous n'avons pas voulu laisser ces valeurs dans le fichier de configuration car il est indispensable d'avoir les mêmes ports chez le serveur et chez les clients. Nous avons donc préféré prendre le risque qu'un autre programme utilise les mêmes ports plutôt qu'un utilisateur change ces valeurs.

6.3.6 Threads lancés pour gérer la communication client-serveur

Voici les différents threads lancés par le client et le serveur.

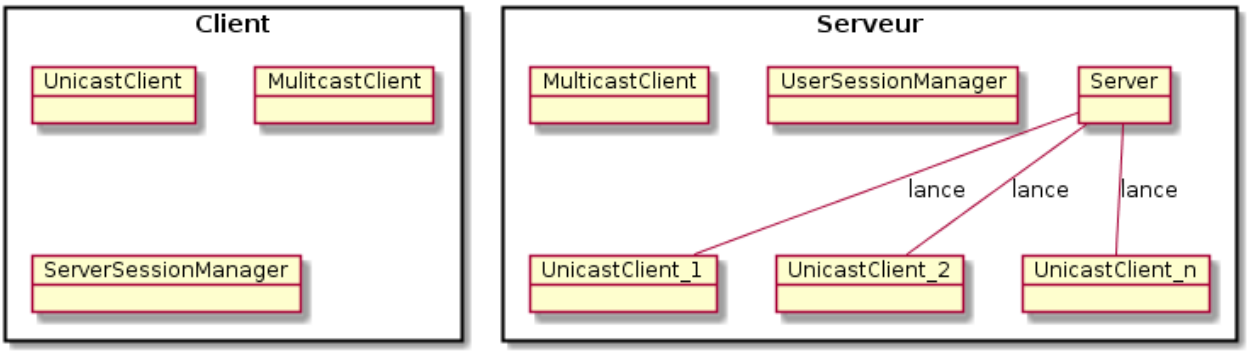

Fig. 13: Threads lancés par les clients et le serveur

6.4 Paquet `session`

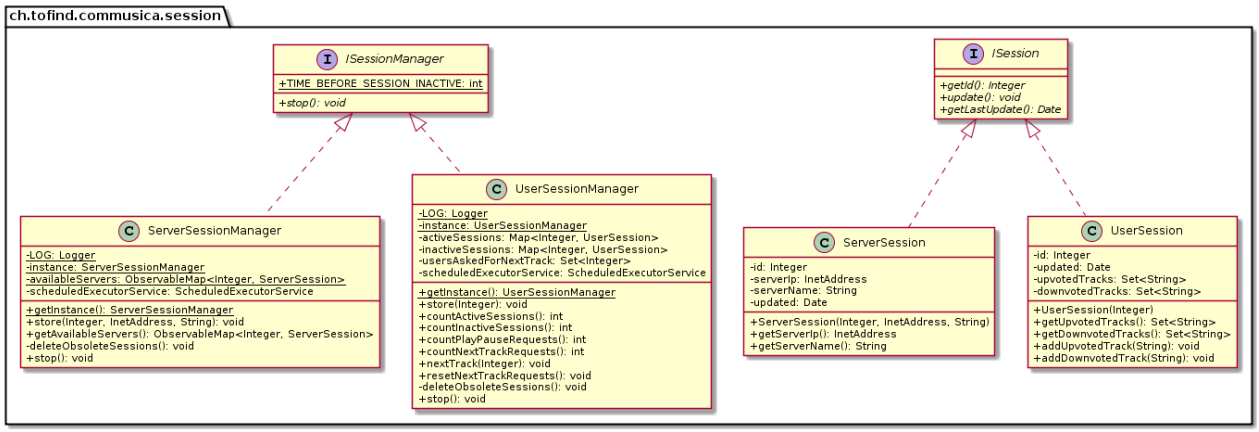

Fig. 14: Paquet `session`

Cette entité permet de gérer la notion de session afin de connaître les personnes connectées et les serveurs accessibles.

6.4.1 `ISession`

L'interface `ISession` permet de définir le fait qu'une session soit caractérisée par un identifiant et une date de dernière mise à jour. Cela est nécessaire pour la bonne exécution de certaines méthodes.

L'identifiant est toujours un ID unique `Integer` qui est généré à l'aide de l'adresse MAC de l'une des interfaces réseau du serveur ou du client.

La méthode `update()` permet de mettre à jour la session afin qu'elle ne soit pas nettoyée par le `ScheduledExecutorService` (voir chapitre 6.4.5).

6.4.2 `ServerSession`

Cette classe représente une session serveur.

En plus de l'identifiant unique, un serveur est représenté par un nom et une adresse de destination afin de savoir comment l'atteindre.

6.4.3 `UserSession`

Cette classe représente une session utilisateur. Le but est de pouvoir identifer un utilisateur unique afin d'éviter qu'il puisse faire plusieurs actions consécutives, sans limite. En plus de cela, cette classe permet de savoir, par exemple, à partir de quand un changement de morceau doit être effectué - la majorité des utilisateurs doit avoir voté un changement de morceau de musique.

6.4.4 `ISessionManager`

Cette interface demande à chaque SessionManager de mettre à disposition une méthode `stop()`. En effet, chaque SessionManager démarre un `ScheduledExecutorService` dont le but est de nettoyer les sessions inactives du système, soit en les supprimant, soit en les désactivant, et ce, à intervalles réguliers définis dans le fichier de configuration (chapitre 6.7.1).

6.4.5 `ServerSessionManager`

Cette classe permet de gérer les différentes sessions des serveurs. Elle permet de stocker et savoir quels sont les serveurs actifs.

Elle est constituée d'une `Map` permettant de stocker les différents serveurs accessibles. Cette liste est mise à jour à chaque fois qu'un serveur envoie une mise à jour de sa liste de lecture à l'aide de la commande `PLAYLIST_UPDATE`.

Elle est nettoyée à l'aide du `ScheduledExecutorService` afin de supprimer de l'interface graphique les serveurs qui ne sont plus accessibles.

6.4.6 `UserSessionManager`

Cette classe permet de gérer le stockage, le nettoyage ainsi que les votes des différentes sessions utilisateur. Elle permet de garder à jour deux listes qui permettent de stocker les utilisateurs actifs et inactifs. Elle a un système de mise à jour automatique à l'aide d'un `ScheduledExecutorService`. Pour ce qui est des votes, elle permet d'empêcher qu'un utilisateur ne vote plusieurs fois pour la même action - score, pause, etc.

6.5 Paquet `media`

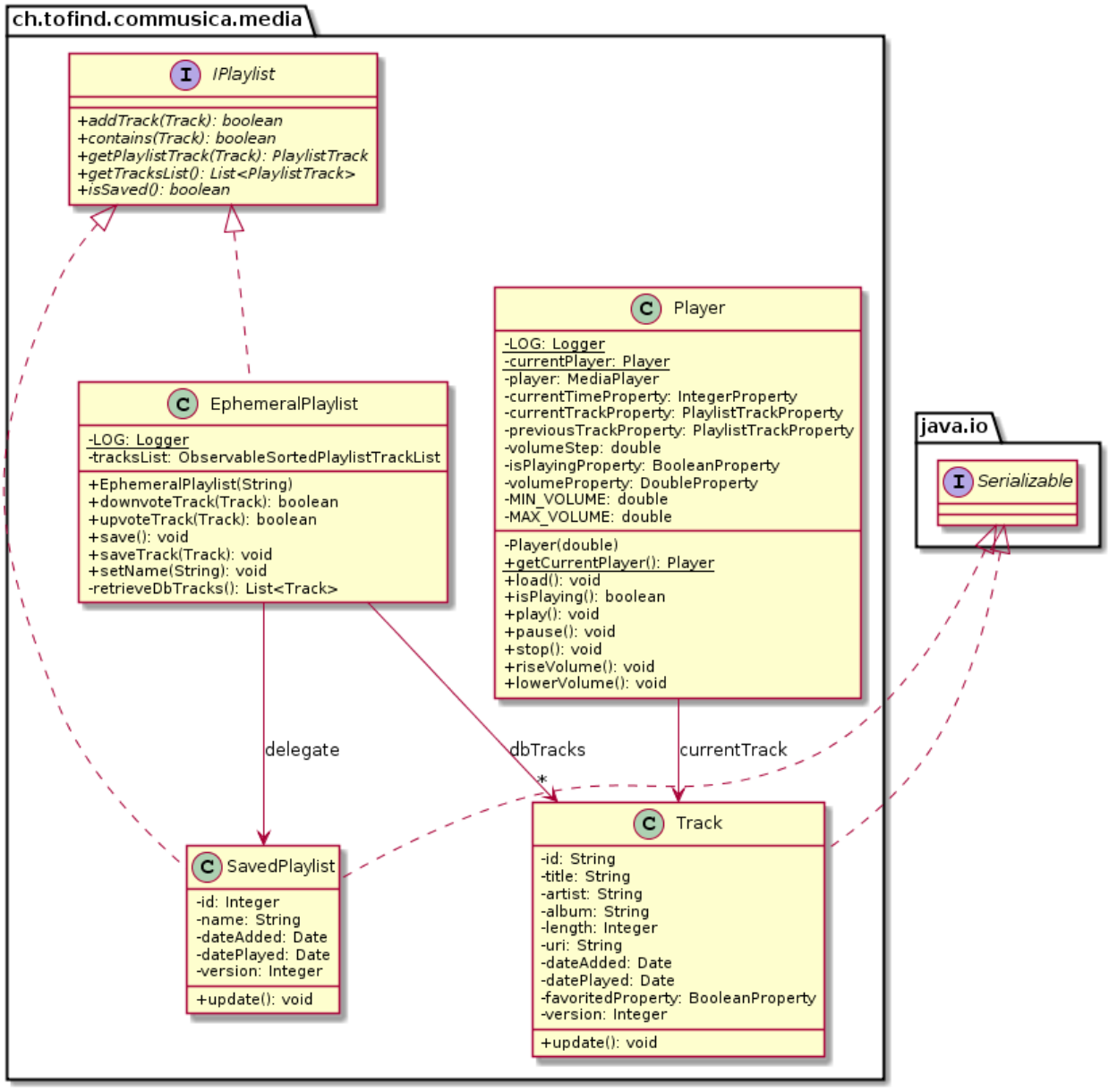

Fig. 15: Paquet `media`

Ce package permet de définir tous les éléments nécessaires à la gestion de la musique au niveau de l'application.

6.5.1 `EphemeralPlaylist`

La classe EphemeralPlaylist représente la liste de lecture en cours de construction et de lecture. Cette classe permet de mettre à jour l'interface graphique, lors d'une action sur un élément de la playlist. La mise à jour se fait grâce au pattern observeur, à travers la liste `ObservableSortedPlaylistTrackList`, qui joue, en même temps, le rôle d'observable et d'observeur. Elle observe les morceaux de musique de la liste dans le but de changer son état en cas de vote positif ou négatif, et est observable dans le cas où elle envoie des notifications lors des mises à jour. Dans cette classe, nous avons aussi le champ `delegate` qui représente la liste de lecture qui sera enregistrée dans la base de données pour le suivi de celle-ci.

6.5.2 `Player`

Cette classe permet de réaliser les actions de base sur la musique telles que la lecture, la mise sur pause, passer au morceau suivant, etc.

Lors de l'implémentation, nous avions le choix entre deux paquets : `javafx.scene.media` et `javax.sound` proposant tous deux différentes classes pour gérer des fichiers audio et les jouer. Nous avons préféré utiliser `javafx.scene.media` pour les raisons suivantes :

- Facilité d'implémentation.
- Accepte plus de formats que `javax.sound`. Par exemple, MP3 n'est pas supporté par `javax.sound`.
- Beaucoup plus abstrait, `javax.sound` demande de travailler directement au niveau des octets.

De plus, `javafx.scene.media` étant issu de JavaFX, que nous utilisons pour l'interface graphique, s'intègre mieux à cette dernière.

Le paquet `javafx.scene.media` propose les classes suivantes :

- `Media` représente une ressource média qui contient des informations telles que la source, la résolution et les métadonnées.
- `MediaPlayer` est le composant clé fournissant les contrôles pour la lecture de médias.
- `MediaView` permet de supporter l'animation, la translucidité et les effets.

Nous avons aussi utilisé, dans cette classe, les propriétés JavaFX dont le but est de mettre à jour de manière automatique l'interface utilisateur lorsqu'une modification se produit.

Une fois que `Player` a fini de lire un morceau, il va interroger le `PlaylistManager` qui va lui fournir le prochain morceau à jouer, en se basant sur le nombre de votes de ce dernier à l'aide du `VoteComparator`.

6.5.3 `SavedPlaylist`

Cette classe est utilisée afin de stocker une `EphermeralPlaylist` dans la base de données et en récupérer une ancienne playlist.

La différence entre les deux classes se situe dans le fait que `EphermeralPlaylist` est une playlist en cours de construction alors que `SavedPlaylist` est le résutat final d'une `EphermeralPlaylist`.

Seule `SavedPlaylist` est stockée dans la base de données et est persistante dans la vie du programme.

6.5.4 `Track`

Cette classe représente un morceau de musique, elle en regroupe toutes les informations nécessaires à une identité unique. Nous avons implémenté trois constructeurs :

- Le constructeur vide : toutes les classes persistantes doivent avoir un constructeur par défaut pour que Hibernate puisse les instancier en utilisant le constructeur `Constructor.newInstance()`.
- **`public`** `Track(String id, String title, String artist, String album, Integer length, String uri)` : constructeur permettant de créer une instance de `Track` lorsque tous les paramètres sont connus.
- **`public`** `Track(AudioFile audioFile)` : constructeur permettant de créer une instance de `Track` à partir d'un fichier audio. Il est utile lorsque nous souhaitons transférer un fichier et effectuer un contrôle sur un morceau de musique, au lieu de vérifier le fichier audio lui-même.

6.6 Paquet `playlist`

Le paquet `playlist` met en oeuvre ce qui a trait à la gestion des playlists, dans notre cas :

- Le lien entre un certain morceau et les playlists dans lesquelles il se trouve.
- La sélection d'une certaine playlist.
- La gestion des votes positifs et négatifs concernant les morceaux contenus dans une liste de lecture spécifique.

6.6.1 `PlaylistManager`

La classe `PlaylistManager` représente un gestionnaire de playlists et a plusieurs utilités :

- Récupérer la liste de lecture en cours de création.
- Récupérer les playlists sauvegardées.
- Récupérer la liste de lecture des favoris.
- Ajouter ou supprimer des morceaux à la liste de lecture des favoris.
- Créer ou supprimer une playlist.

6.6.2 `PlaylistTrack`

La classe `PlaylistTrack` permet non seulement de représenter le lien entre une morceau et une playlist, mais aussi de connaître le nombre de votes du morceau, ce qui sera ensuite utile au niveau de la classe `VoteComparator` qui organise les morceaux dans la liste de lecture selon le nombre de votes. Cela peut être fait grâce au fait que `PlaylistTrack` met à disposition une variable `votesProperty` à laquelle un observeur a été ajouté afin que l'interface graphique se réorganise correctement.

6.6.3 `PlaylistTrackId`

Cette classe permet de créer le lien entre une certaine liste de lecture et un morceau. Grâce à l'implémentation d'un hashcode, nous pouvons nous servir de celui-ci afin de vérifier que le morceau relié à la liste de lecture n'existe pas déjà dans la base de données.

6.6.4 `VoteComparator`

Le comparateur de vote ne possède qu'une fonction. Celle-ci sert tout simplement à déterminer, entre deux morceaux, lequel a le plus grand nombre de votes. Il a été créé dans le but de réorganiser la liste de lecture en commençant par les morceaux les plus votés.

6.7 Paquet `utils`

Le paquet `utils` réunit tous les utilitaires dont nous avons eu besoin au sein de plusieurs classes et dont l'implémentation n'avait aucun sens au sein desdites classes. L'utilité de chaque classe diffère alors énormément.

6.7.1 `Configuration`

Cette classe permet la récupération des propriétés définies dans un fichier de configuration. Elle fixe le fichier de configuration à utiliser et permet l'accès aux propriétés.

Le ficher de configuration du programme utilisé dans le cadre de ce projet est `commusica.properties`. Il donne accès aux paramètres suivants :

- DEBUG : permet de choisir d'afficher ou non la sortie du programme
- DATE_FORMAT : choix du format de la date
- VOLUME_STEP : choix du pas d'augmentation et abaissement de la musique
- TRACKS_DIRECTORY : choix du chemin relatif où les morceaux seront stockés
- TIME_BEFORE_SESSION_INACTIVE : choix du délai d'inactivité d'une session
- TIME_BETWEEN_PLAYLIST_UPDATES : choix du délai de mise à jour des playlists et leurs morceaux

6.7.2 `Network`

Cette classe permet de récupérer toutes les informations basiques de la machine concernant le réseau. Elle va, en outre, permettre de récupérer les interfaces disponibles nécessaires à la connexion à un certain serveur et configurer le réseau pour le reste de l'application.

6.7.3 `ObservableSortedPlaylistTracklist`

Cette classe permet de récupérer les informations nécessaires à l'affichage des morceaux de musique dans la liste de lecture en écoute. Cet utilitaire a été créé afin de pouvoir faciliter la récupération d'informations depuis les classes implémentant l'interface graphique.

6.7.4 `Serialize`

Grâce à la librairie Gson de Google, cette classe est utilisée dans la sérialisation (Objet Java -> JSON) et désérialisation (JSON -> Objet Java).

6.7.5 `EphemeralPlaylistSerializer`

Cette classe permet de sérialiser et désérialiser une liste de lecture en JSON. Son utilité réside principalement dans la communication réseau.

6.7.6 `Logger`

Cette classe a été créée uniquement pour aider à déboguer le programme et comprendre ce qu'il se passe à chaque étape. Son affichage permet de savoir dans quelle classe a lieu une action. Des couleurs ont été attribuées aux différentes notifications :

- Bleu, pour les informations
- Rouge, pour les erreurs
- Vert, pour les succès
- Jaune, pour les avertissements

L'affichage des logs peut être désactivé au niveau du fichier de configuration `commusica.properties` en réglant la valeur de `DEBUG` à 0.

6.8 Paquet `core`

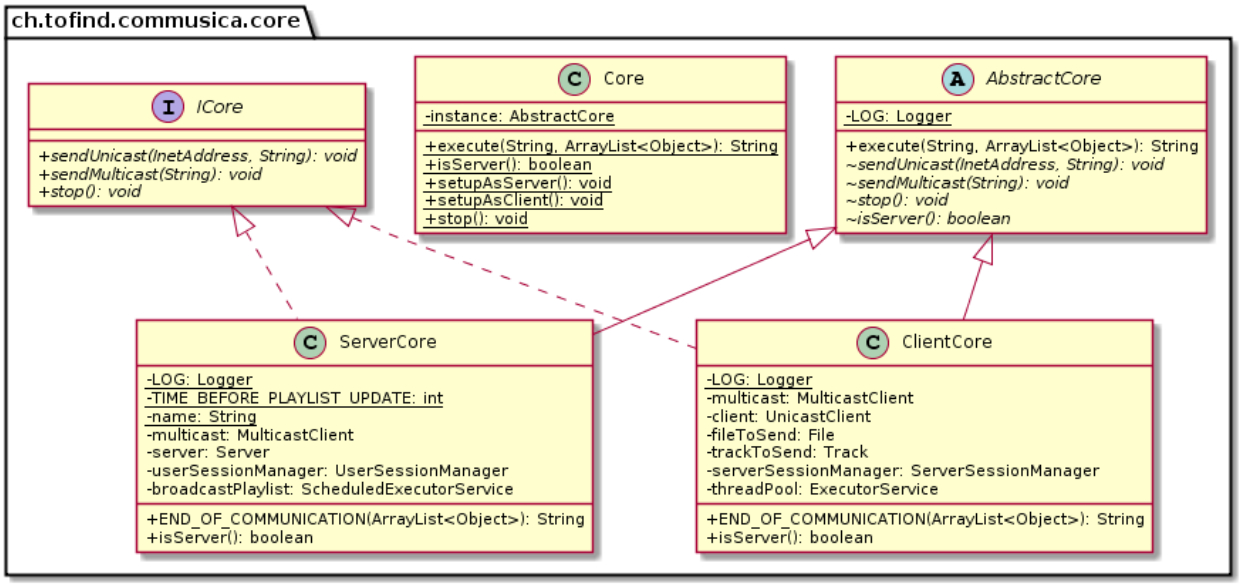

FIG. 16: Paquet `core`

Pour garder un niveau d'abstraction le plus élevé possible, nous avons voulu faire transiter, à travers un contrôleur, toutes les informations venant du réseau et des utilisateurs, le but étant d'avoir le même point d'entrée que l'on soit client ou serveur. Pour cela, il nous fallait un contrôleur central qui puisse être appelé de la même façon, quel que soit le choix de l'identité. C'est alors à celui-ci de vérifier l'existence d'une fonction et de communiquer l'action à exécuter à l'entité concernée. Pour résoudre ce problème, notre raisonnement nous a mené à nous tourner vers la réflexivité offerte par Java. Ce mécanisme permet d'instancier des méthodes à l'exécution en utilisant la méthode `invoke(Object obj, Object... args)` ayant comme premier paramètre un String représentant le nom de la méthode à invoquer et comme deuxième paramètre un tableau d'`Object` contentant les différents arguments dont la méthode invoquée a besoin.

Il nous fallait maintenant une classe qui puisse jouer le rôle du contrôleur. Pour cela, nous avons développé les `Core`, qui se trouvent dans le paquet `core`.

6.8.1 `Core`

C'est une classe statique qui joue le rôle de point d'entrée. Elle dispose d'un attribut `AbstractCore` qui sera instancié soit en `ClientCore`, soit en `ServerCore`. Elle met aussi à disposition des méthodes statiques qui seront atteignables depuis les autres classes du programme.

En plus des méthodes lui permettant de se paramétrer comme client ou serveur, la plus importante est la suivante :

```
1 public static String execute(String command, ArrayList<Object> args)
```

Cette méthode statique qui peut être appelée n'importe où dans le programme appellera la méthode du même nom de la classe `AbstractCore`, qui est décrite au chapitre 6.8.3. Cela permet de pouvoir exécuter des commandes quelque soit le type de Core configuré : serveur ou client.

6.8.2 `ICore`

`ICore` est une interface qui définit ce qui est nécessaire au Core, à savoir des méthodes permettant d'envoyer des messages en Unicast ou en Multicast et une méthode pour arrêter le Core. Toutes les classes héritant d'`AbstractCore` doivent implémenter cette interface.

6.8.3 AbstractCore

Cette classe abstraite met à disposition les méthodes permettant à ses sous-classes de s'exécuter correctement. Contrairement à `Core`, cette classe va utiliser la réflexivité dans sa méthode execute(), comme ceci :

```
1  public synchronized String execute(String command, ArrayList<Object> args) {
2
3      String result = "";
4
5      try {
6          Method method = this.getClass().getMethod( command,
               ArrayList.class);
7          result = (String) method.invoke(this, args);
8      } catch (NoSuchMethodException e) {
9          // Do nothing
10     } catch (IllegalAccessException | InvocationTargetException e) {
11         LOG.error(e);
12     }
13
14     return result;
15 }
```

Nous recevons une commande et un tableau correspondant aux arguments de la méthode à invoquer. Ensuite, le programme essaie de trouver la méthode ayant un nom correspondant à la commande. Si c'est le cas, cette dernière va l'invoquer et donc exécuter ladite méthode, sinon une exception est levée. C'est grâce à cette méthode que tout prend son sens, car on a maintenant une instance d'`AbstractCore` qui est soit `ClientCore`, soit `ServerCore` avec une seule méthode pour en appeler d'autres qui seront, elles, implémentées dans les sous-classes d'`AbstractCore`.

6.8.4 ServerCore et ClientCore

Ces deux classes héritant d'`AbstractCore` et implémentant `ICore` sont les classes les plus importantes du projet. C'est ici que la majorité des actions - transfert de la musique, action à effectuer lors d'un appui sur un bouton, etc. - se fera. Lors de l'envoi des commandes, ces classes fonctionnent avec un système d'états dans lequel ces derniers peuvent être changés en recevant des commandes depuis le réseau ou depuis le code. Elles ont une forte interaction avec les classes s'occupant des échanges réseau puisque c'est ici que toutes les informations reçues depuis le réseau vont passer. Grâce à la réflexivité offerte par l'`AbstractCore`, il est donc extrêmement facile de définir de nouvelles méthodes dans ces classes. Pour cela, il faut déclarer une méthode portant le nom d'une commande.

Grâce à ces classes, nous avons réglé le problème de contrôleur central par lequel tout transitera.

6.8.5 Envoi d'un fichier audio d'un client vers un serveur

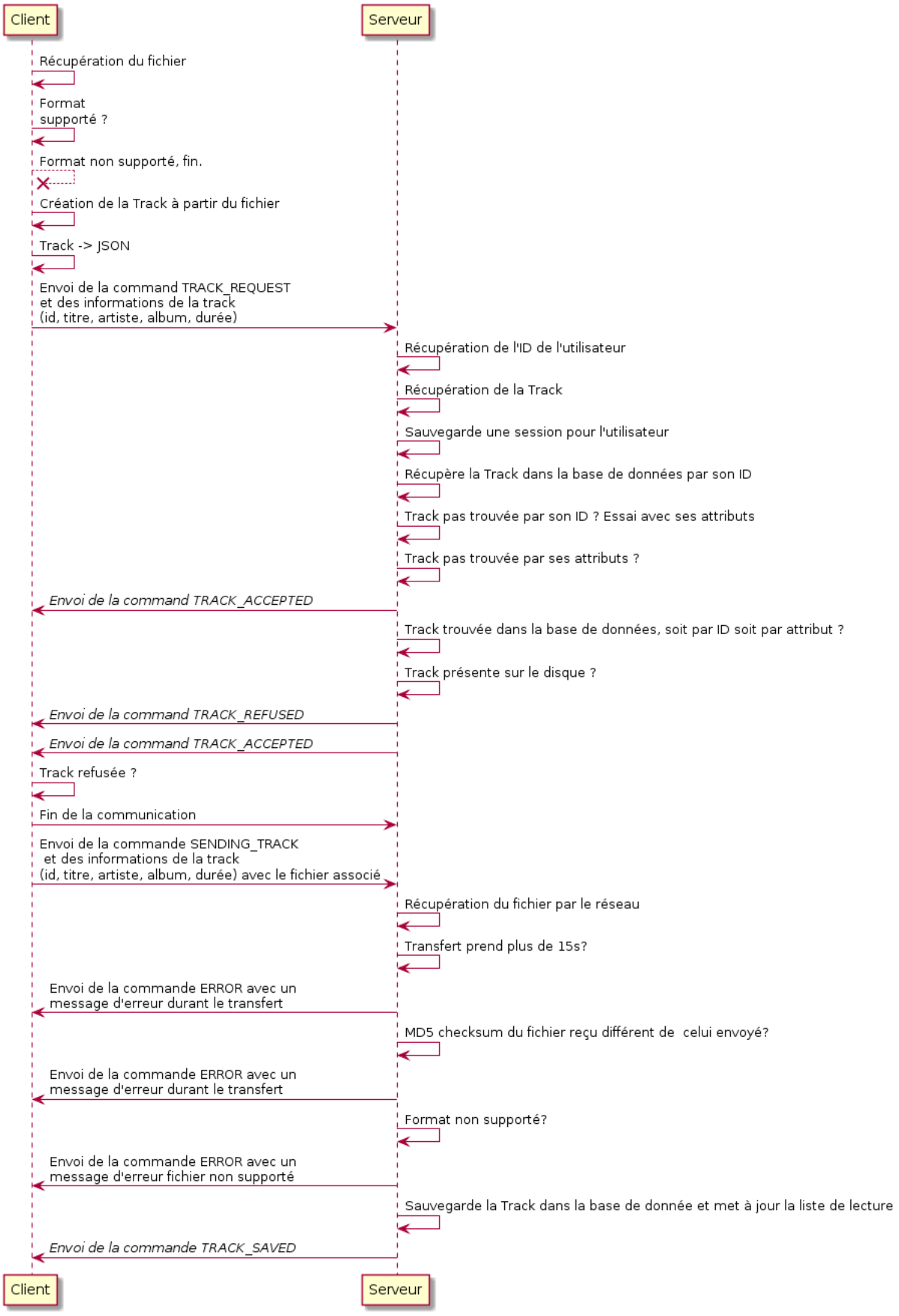

FIG. 17: Diagramme de séquence de l'envoi d'un fichier audio

6.9 Paquet et ressources `ui`

Concernant l'interface graphique, nous avons utilisé la librairie JavaFX. Celle-ci nous a permis de faire usage de l'outil Scene Builder afin de développer, en premier lieu, une maquette qui s'est ensuite développée, à travers plusieurs étapes, en l'interface graphique d'aujourd'hui. Le fonctionnement de JavaFX demande à avoir deux notions qui communiquent entre elles : un ou plusieurs fichiers FXML qui définissent l'arrangement de la fenêtre et une ou plusieurs classes Java qui permettent de lancer la fenêtre et communiquer avec ses composants. Dans un premier temps, nous avons développé un fichier FXML avec l'aide de Scene Builder. Grâce à celui-ci, nous avons pu apprendre les bons usages FXML. Nous avons ensuite créé un fichier Java depuis lequel nous étions capables lancer la fenêtre au démarrage du programme. Cependant, le code se développant et devenant de plus en plus important, nous avons pris la décision de diviser aussi bien les fichiers FXML que les fichiers Java en plusieurs classes permettant d'avoir un regard plus précis sur chaque partie de notre implémentation. Ainsi, nous avons aujourd'hui plusieurs classes Java et plusieurs fichiers FXML qui sont reliés à leur classe principale `UIController` respectivement `main.fxml`.

La description des classes se fera selon l'ordre des vues dans l'interface graphique, en partant de la vue en haut à gauche pour finir par la vue en bas au centre. Nous allons tout d'abord commencer par la fenêtre de configuration apparaissant au lancement du programme, pour continuer avec le contrôleur. Le reste des classes sera ensuite abordé.

6.9.1 `ClientServerDialog`

`ClientServerDialog` est la première fenêtre lancée par le programme. Son lancement se passe alors dans la classe principale `Commusica`. Cette fenêtre permettra tout simplement de choisir entre deux rôles : celui du serveur ou de l'utilisateur lambda. Le choix sera communiqué au Core qui prendra connaissance de la décision, configurera le programme et exécutera le lancement de l'interface graphique appropriée.

Si l'utilisateur a choisi le rôle de serveur, une nouvelle fenêtre apparaîtra afin qu'il puisse choisir le nom du serveur. Ce dernier sera aussi utilisé comme nom de la playlist qui sera diffusée. Après cela, la classe lancera l'`UIController`. Dans le cas où l'utilisateur ne répond pas à la question posée dans la fenêtre de dialogue et la ferme, le programme s'arrête.

6.9.2 `UIController`

`UIController` est la classe qui permet de lier le reste des classes graphiques entre elles. Elle va, en premier lieu, mettre à jour la partie en haut à gauche contenant les playlists et la liste de lecture sélectionnée sera par défaut celle en cours de construction. Mise à part la configuration initiale de la fenêtre, `UIController` permet aussi toutes les actions basiques de l'interface graphique :

- Afficher des alertes
- Obtenir la liste de lecture actuellement visualisée
- Mettre à jour et afficher les playlists
- Fermer la fenêtre proprement lorsque l'utilisateur décide d'arrêter le programme

`UIController` va tout simplement faire appel aux différentes classes du paquet `ui` afin de s'informer de l'état de chaque partie composant l'UI lors d'une demande depuis l'extérieur.

6.9.3 `PlaylistsListView`

`PlaylistsListView` concerne la vue en haut à gauche affichant les playlists disponibles :

- PLAYING : la liste de lecture en cours de création
- FAVORITES : la liste de lecture des favoris
- SAVED : la liste des playlists sauvegardées d'anciens événements

Comme spécifié au chapitre précédent, la liste sélectionnée par défaut est la liste en cours de création. Dans la classe `PlaylistsListView`, nous faisons usage d'une méthode de la classe `FXCollections` permettant d'attacher un observeur à n'importe quel objet du programme. Ainsi, nous pouvons facilement modifier l'affichage des playlists au fur et à mesure des actions faites au niveau du serveur ou du client.

6.9.4 `TracksListView`

La classe `TracksListView` agit sur le panneau en haut au centre de l'interface graphique principale en le dessinant et définissant les usages basiques de celui-ci. Cette classe permet d'afficher une liste de lecture et glisser/déposer un élément audio au sein du panneau, grâce à la méthode `initializeDragAndDrop()`. La liste de lecture est initialisée comme liste observable, ce qui fait que dès qu'un changement subvient, celle-ci se met à jour. Cependant, elle n'est pas encore peuplée par les morceaux de musique contenus dans la playlist. Cette question ainsi que celle des upvotes, downvotes et favoris sont traitées dans une autre classe implémentée spécialement pour cet usage : `PlaylistTrackCell`.

La méthode `initializeDragAndDrop()` de la classe `TracksListView` mérite une explication plus détaillée. Nous avons longtemps réfléchi à la meilleure façon d'implémenter le téléchargement d'un morceau. Le "drag and drop" (glisser/déposer) nous a finalement semblé être la technique la plus intuitive d'ajout de musique. Cette méthode, relativement complexe, nous permet donc de déterminer quand une personne a déposé un fichier dans le panneau et ce, grâce à la méthode JavaFX `setOnDragDropped()` de la classe `TransferMode`. C'est alors que nous allons faire usage du constructeur de la classe `Track` prenant en paramètre un `AudioFile`. Si c'est le serveur qui a glissé/déposé un morceau, alors la méthode appellera directement la méthode du `PlaylistManager` permettant d'ajouter un morceau. Dans le cas du client, la méthode passera d'abord par la classe `Core` à laquelle il enverra la commande `SEND_TRACK_REQUEST` avec comme argument le morceau sérialisé en JSON. Nous remarquons ici, encore une fois, l'intérêt et l'importance de la classe `Core`.

6.9.5 `PlaylistTrackCell`

`PlaylistTrackCell` est une classe utilisée dans chaque cellule de la liste centrale afin de définir les boutons d'upvote, downvote et favoris et leurs actions. Elle va également permettre de définir le titre, l'artiste, l'album et le nombre de votes d'un morceau. Concernant les votes, deux fonctions - une pour les votes positifs et l'autre pour les votes négatifs - permettent de communiquer avec le `Core` à travers des commandes. Les commandes - `SEND_DOWNVOTE_TRACK_REQUEST` et `SEND_UPVOTE_TRACK_REQUEST` - sont utilisées dans ces deux cas spécifiques car l'incidence qu'aura un vote sera globale à tous les participants. Ainsi, le `Core` doit être averti du fait que l'événement a eu lieu pour en informer le serveur afin qu'il renvoie l'information à tout le monde. Encore une fois, le `Core` use de son pouvoir de messager à travers le programme. Dans le cas des favoris, il n'y a nul besoin de passer par le `Core` car tout ce que l'utilisateur veut, c'est enregistrer l'information dans sa liste personnelle de morceaux de musique favoris.

6.9.6 `SettingsView`

Le panneau en haut à droite de l'interface graphique propose quelques réglages simples mais importants quant au bon fonctionnement du programme. Ce panneau n'a pas le même comportement selon si l'utilisateur est un client ou un serveur.

Si l'utilisateur a choisi d'être le serveur, alors le nom qu'il aura choisi lors du lancement de l'application sera affiché afin qu'il lui soit simple de retrouver cette information à tout moment. Ce champ est final et ne peut pas être modifié tant que le programme est en cours d'exécution.

Si l'utilisateur a choisi d'être un client, une liste des serveurs disponibles sur le réseau local s'affichera. Lorsqu'un serveur est sélectionné dans la liste, le programme se charge de connecter le client au serveur précédemment choisi.

Enfin, un deuxième paramètre est commun aux deux types d'utilisateurs : le choix de l'interface réseau à utiliser. Celle-ci se choisit à l'aide d'un menu déroulant qui nous fournit la liste des interfaces réseau disponibles ainsi que leur adresse IPv4 associée.

6.9.7 `PreviousTrackView`

Dans le panneau en bas à gauche, nous pouvons apercevoir un espace réservé au morceau qui vient de se terminer. Ce panneau nous a semblé utile de par le fait que, souvent, nous nous sommes personnellement retrouvés à vouloir noter le nom d'un morceau que nous venions d'écouter et, le temps de prendre notre téléphone pour identifier ledit morceau de musique, celui-ci avait eu le temps de se terminer. Ainsi, ce panneau offre la possibilité à tous les utilisateurs de retrouver facilement et sauvegarder en un seul *clic* les informations d'un morceau. Pour ce panneau, nous n'avons pas repris le même type de cellule que dans le panneau central du haut, car il n'y a pas de sens au fait de pouvoir upvoter ou downvoter un morceau déjà écouté. C'est pourquoi, nous avons créé

un panneau sur mesure contenant uniquement l'étoile des favoris et permettant ainsi uniquement d'ajouter le morceau de musique à sa liste de morceaux favoris.

6.9.8 `CurrentTrackView`

Dans le panneau du bas, au milieu, nous pouvons apercevoir le résumé du morceau actuellement en écoute. Les boutons ainsi que les informations sont exactement les mêmes que dans le cas du dernier morceau grisé, affiché dans la liste de lecture du panneau en haut au centre. Nous avons choisi cet affichage de façon à pouvoir faciliter l'accès au morceau actuel dans le cas où l'utilisateur aurait décidé de faire défiler la liste de lecture et aurait perdu de vue le morceau actuel. Le vrai défi de cette classe a cependant été celui de pouvoir remplir la jauge d'écoute selon l'avancement du morceau de musique. Cela a évidemment été fait à travers un observeur sur l'instance de `Player` qui possède l'information sur le temps écoulé. Sur la gauche du panneau central, nous pouvons également apercevoir des boutons de contrôle.

6.9.8.1 `PlayerControlsView`

`PlayerControlsView`, qui se trouve dans le même panneau que `CurrentTrackView`représente les boutons *play/pause*, *morceau suivant*, *morceau précédent* et *volume*. Ces quatre boutons représentent en fait cinq actions distinctes qui transiteront toutes à travers le `Core`. En effet, nous nous trouvons encore une fois face à une classe de l'interface graphique dont le `Core` est indispensable à son bon fonctionnement. Le `Core` est en mesure de déterminer vers qui il devra tourner la demande d'action à travers l'une des commandes suivantes :

- `SEND_TURN_VOLUME_DOWN_REQUEST` : pour baisser le volume
- `SEND_TURN_VOLUME_UP_REQUEST` : pour augmenter le volume
- `SEND_NEXT_TRACK_REQUEST` : pour écouter le morceau suivant
- `SEND_PREVIOUS_TRACK_REQUEST` : pour écouter le morceau précédente
- `SEND_PLAY_PAUSE_REQUEST` : pour arrêter ou démarrer la musique

Dans ces cas précis, c'est la classe `UserSessionManager`qui sera concernée par la commande. Finalement, nous voyons dans cette classe encore une trace de ce que nous avions initialement implémenté. En effet, comme dans tous controleurs de musique, les boutons play/pause, morceau suivant et morceau précédent sont toujours présents. Cependant, dans le concept que nous visions à créer, nous n'avons jamais voulu permettre aux utilisateurs de revenir en arrière mais bien de se trouver dans un flux continu de musique.

6.9.9 Fichiers FXML

Le fichier FXML de base, tout comme le fichier Java de base, a été découpé en plusieurs fichiers afin de comprendre plus facilement l'implémentation et la modifier. Comme dans les fichiers Java, nous avons un fichier principal, `main.fxml`, qui permet de découper l'interface principale en plusieurs panneaux. Ensuite, pour chacun des panneaux, nous avons des fichiers portant le même nom que leurs classes Java. Nous avons dû ajouter aux fichiers FXML un ID à chaque structure dont les actions nous intéressaient, et, pour certaines, nous avons également dû lier une action à une certaine méthode. Ainsi, dès l'instant que, dans le code Java, nous rencontrons un **`@FXML`**, cela veut dire que nous avons un lien direct avec les structures des fichiers FXML.

7 Technologies utilisées

7.1 Singleton

Comme expliqué précédemment, nous avons implémenté une partie de nos classes comme étant des Singleton. Ce patron de conception fait en sorte qu'une classe n'ait qu'une seule instance.
Voici un exemple avec la classe `FileManager`:

```
1  private static FileManager instance = null;
2
3  private FileManager() {}
4
5  public static FileManager getInstance() {
6
7      if (instance == null) {
8          synchronized (FileManager.class) {
9              if (instance == null) {
10                 instance = new FileManager();
11             }
12         }
13     }
14
15     return instance;
16 }
```

Ce patron de conception est simple à mettre en oeuvre avec une variable d'instance de la classe, une constructeur privé et une méthode `getInstance()` renvoyant l'instance de la classe. Il est particulièrement adapté à toutes nos classes de type `Manager`, car il ne doit y avoir qu'une seule instance de celle-ci par programme.

7.2 Réflexion

La réflexion offerte par Java a été utilisée dans le cadre des Core afin de pouvoir déterminer, lors de l'exécution, si une méthode était implémentée et pouvait être appelée. Cela a permis d'éviter la redondance de code entre un Core côté serveur et un Core côté client et n'implémenter que les méthodes qui devaient être gérées par l'une ou l'autre de ces entités.

7.3 ThreadPool

`ThreadPool` est un outil offert par Java permettant de définir un objet `ExecutorService` mettant à disposition un nombre de threads défini par le programmeur. Il suffit ensuite de lui soumettre des objets instance de la classe `Thread` pour qu'il les lance automatiquement selon la disponibilité de son *pool*. Cela permet donc de limiter le nombre de threads lancés simultanément.
Voici un exemple d'utilisation pour le lancement d'un nouveau thread, lors de la connexion d'un client sur le socket du serveur :

```
1  Socket clientSocket = socket.accept();
2
3  Thread client = new Thread(new UnicastClient(clientSocket));
4  threadPool.submit(client);
```

7.4 ScheduledExecutorService

Sous-classe de `ExecutorService`, cette classe permet l'exécution d'un bloc de code à une fréquence définie par le programmeur. Nous l'avons utilisée pour des tâches telles que l'envoi de la liste de lecture ou la suppression de sessions utilisateurs obsolètes.

Exemple de l'envoi de la liste de lecture toutes les `TIME_BEFORE_PLAYLIST_UPDATE` secondes :

```
broadcastPlaylist = Executors.newScheduledThreadPool(1);
broadcastPlaylist.scheduleAtFixedRate(() -> {
    execute(ApplicationProtocol.SEND_PLAYLIST_UPDATE, null);
}, 0, TIME_BEFORE_PLAYLIST_UPDATE, TimeUnit.SECONDS);
```

7.5 Gson

Gson est une librairie développée par Google permettant la sérialisation et la désérialisation d'objets en JSON. Nous l'avons utilisé principalement pour envoyer les différents objets à travers le réseau. Nous avons choisi cette libraire car nous l'avons déjà utilisée en cours et qu'elle permet en peu de lignes d'avoir une sérialisation ou désérialisation rapide.

7.6 Hibernate

Hibernate est un outil ORM (Object Relational Mapping) qui simplifie la création et l'interaction avec la base de données. Il offre la possibilité de créer automatiquement les tables de la base de données en se basant sur les objets Java. Il n'est donc pas nécessaire de les créer manuellement et il gère la sauvegarde et l'intégrité de la base de données par lui-même.

Nous l'avons utilisé afin de pouvoir nous concentrer sur le code et lui déléguer la gestion de la base de données.

7.7 JavaFX

JavaFX est une bibliothèque Java permettant la création d'applications de bureau. Les applications écrites à l'aide de cette bibliothèque peuvent fonctionner sur plusieurs plateformes. Par rapport à Swing, JavaFX nous offre les avantages suivants :

- Les styles CSS, qui nous ont permis de styliser notre application
- La séparation de l'interface et du code

Le point majeur qui nous a fait choisir JavaFX est la séparation facile entre le code et l'interface graphique, à l'aide de fichiers FXML, permettant de définir la structure de notre interface sans polluer le code avec des concepts qui n'auraient pas de sens dans celui-ci.

7.8 JAudiotagger

JAudiotagger est une API Java pour la lecture et l'écriture des métadonnées des fichiers audio. Il supporte des formats tels que MP3, MP4, WAV, etc.

Nous l'avons utilisé dans notre projet car JavaFX ne propose pas d'outil de récupération des métadonnées des morceaux de musique et car c'est la librairie la plus connue et adaptée à nos besoins.

7.9 Capsule

Capsule est un outil de déploiement pour les applications Java. Une capsule est un fichier JAR exécutable unique qui contient tout ce que l'application doit avoir pour pouvoir s'exécuter.

Nous l'avons utilisé dans le but de forcer l'utilisation de l'IPv4 pour les interfaces réseaux sur les ordinateurs Apple qui, par défaut, préfèrent l'IPv6 et ne peuvent donc pas communiquer avec les autres systèmes qui, eux, préfèrent l'IPv4.

Lors du lancement de notre programme, la réelle exécution de celui-ci est la suivante :

1. Exécution du programme Commusica
2. Lancement de Capsule
3. Définition des paramètres à utiliser pour la JVM 2
4. Lancement de Commusica avec les paramètres JVM souhaités
5. Démarrage de Commusica‘

7.10 Git / GitHub

Git est un outil de gestion de versions qui permet de simplifier le développement d'une application en gérant automatiquement la fusion de code de deux auteurs différents et, permet également de pouvoir avoir un historique des actions effectuées tout au long du projet.

Nous l'avons utilisé afin de permettre à chacun de développer séparément et qu'il puisse gérer la fusion automatiquement. Nous pouvions, au besoin, effectuer des tests sans mettre en péril le reste du projet à l'aide de branches. Nous avons utilisé GitHub afin de centraliser cela sur Internet.

7.11 IntelliJ IDEA

IntelliJ IDEA est un environnement de développement intégré, autrement dit, un ensemble d'outils destinés au développement logiciel.

Nous l'avons choisi pour les raisons suivantes :

- Intégration avec les outils de gestions de versions tels que Git.
- Autocomplétion hors pair.
- Analyse et inspection : il analyse en temps réel et en permanence le code, à la recherche de problèmes potentiels.
- Débogueur simple et efficace à utiliser.

7.12 Apache Maven

Apache Maven est un outil de gestion de projet basé sur POM (Modèle d'Objet de Projet).

Nous l'avons utilisé dans le cadre de notre projet afin de pouvoir gérer les dépendances et la compilation de façon unifiées chez tous les développeurs. Il nous a permis de définir une librairie et sa version à utiliser et ainsi, le code de tous les développeurs se base sur les mêmes versions et compile de la même façon pour s'assurer du bon fonctionnement du programme.

7.13 Scene Builder

Scene builder est un outil qui permet de créer des fichiers au formats FXML via un éditeur graphique.

Nous l'avons utilisé afin de découvrir les principes de réalisation d'interfaces graphiques à l'aide de JavaFX.

7.14 Wireshark

Wireshark est un outil essentiel pour comprendre les mécanismes de fonctionnement des protocoles de communication sur les réseaux. Il capture des paquets directement sur les interfaces réseau du système utilisé.

Nous l'avons utilisé dans notre projet afin de vérifier que la communication réseau entre le client et le serveur marche comme prévu.

7.15 PlantUML

PlantUML est un outil gratuit et open source qui permet la génération de schémas UML de toutes sortes - diagrammes de classe, diagrammes de séquences, diagrammes d'activités, etc. - et ce, à l'aide de fichiers textes.

Il a été utilisé afin de pouvoir très simplement créer des schémas UML qui pouvaient être améliorés par plusieurs personnes en même temps à l'aide de Git grâce au fait que ce sont simplement des fichiers textes.

8 Tests réalisés

	Test	Résultat	Remarques
Client	L'application se lance en tant que client	Oui	
	Les clients voient les serveurs disponibles	Oui	Certaines fois, un redémarage de la carte réseau est nécessaire
	Un client peut se connecter à un serveur	Oui	
	Plusieurs clients peuvent se connecter à un serveur	Oui	
	Les clients voient la mise à jour de la liste de lecture	Oui	
	Les clients peuvent upvoter/downvoter les morceaux	Oui	
	Les client peuvent proposer de nouveaux morceaux	Oui	
	Un client peut passer d'un serveur à l'autre	Partiel	Fonctionne mais l'interface graphique ne se met pas à jour
	Un client démarré avant le serveur voit ce dernier lorsque celui-ci démarre	Oui	
	Un client peut se déconnecter du serveur sans que cela aie une quelconque incidence	Oui	
Commun	Un utilisateur ne peut pas changer le score d'un morceau de plus d'un point	Oui	
	Une action est effectuée lorsque la majorité des utilisateurs ont voté pour cette action	Oui	
	Un morceau ne peut pas être 2x dans la liste de lecture	Oui	
	Les scores font changer l'ordre de la liste	Oui	
	Plusieurs personnes peuvent envoyer des fichiers en même temps	Oui	
	Un morceau peut être favorisé et mis dans la playlist "Favorites" par un client	Oui	
	La base de données est nettoyée correctement	Oui	
	Le système de fichiers est nettoyé correctement	Oui	
	La liste des interfaces est correcte	Oui	
	Changer d'interface influe sur la connexion	Oui	
	Lorsque le serveur se déconnecte, il n'y a aucun problème chez les client	Non	Cela lance des NullPointerException
	Les différentes fenêtres se ferment en appyuant sur la croix rouge	Non	La première fenêtre de choix client/serveur ne se ferme pas
	L'envoi d'un fichier non supporté ne fonctionne pas	Oui	
	Redimensionner la fenêtre fonctionne	Oui	
	Il est possible de voir ses playlist enregistrées	Non	Marche chez le client mais pas chez le serveur
	Tous les fichier dans le bon format sont joués	Non	Certains .mp3 ne se jouent pas
Serveur	L'application se lance en tant que serveur et nous pouvons choisir un nom de serveur	Oui	
	Le serveur peut upvoter/downvoter les morceaux	Oui	
	Le serveur peut changer l'état de la musique	Oui	
	Le serveur peut proposer de nouveaux morceaux	Oui	
	Un morceau peut être favorisé et mis dans la playlist "Favorites" par un serveur	Non	
	Deux serveurs ayant le même nom ne pose pas problème	Non	Problème de mise à jour de l'interface graphique chez les clients
	Les sessions des clients sont bien mises à jour sur le serveur	Oui	
	Un serveur ne peut pas avoir un nom vide	Oui	Il peut par contre avoir un nom composé uniquement d'espaces/tabulations

FIG. 18: Tests réalisés

8.1 Problèmes subsistants

- Il n'y pas moyen de proposer à nouveau un morceau qui a déjà été joué durant la soirée.
- Des exceptions sont levées dans certains cas d'utilisation.

8.2 Problèmes potentiels non testés

- Risque de bloquer toute l'application en cas de charge élevée car la méthode `execute` des Cores est en exclusion mutuelle et donc, peut potentiellement bloquer l'interaction avec le serveur, s'il y a beaucoup de clients connectés et interagissant avec le serveur.

9 Retour sur le cahier des charges

Avec les tests décrits au chapitre 8 et selon notre cahier des charges, voici le récapitulatif des fonctionnalités implémentées dans notre projet.

Fonction	Fonctionnalité importante?	Réalisé?	Remarque
Démarrage et arrêt corrects du programme	Oui	Oui	-
Droit client-serveur	Oui	Oui	-
Notification des actions	Oui	Partiellement	Les actions sont bien transmises au client mais ne sont visibles que dans les logs. Il faut encore lier à l'interface
Paramétrages basiques du serveur	Oui	Oui	-
Effectuer une annonce de connexion	Oui	Oui	L'état du serveur est envoyé à intervals réguliers.
Réception de la musique	Oui	Oui	-
Lecture des fichiers MP3 et M4A	Oui	Oui	-
Ajout de la musique à la base de donnée/système de stockage	Oui	Oui	-
Actions de base sur la musique côté serveur et client	Oui	Oui	Le bouton pour revenir en arrière n'est pas implémenté car inutile dans notre cas. Il n'est présent que par soucis d'esthétisme.
Interface utilisateur	Oui	Oui	-
Contrôle du volume de la musique côté server et client	Oui	Oui	-
Accepter ou refuser l'ajout de nouveaux morceaux	Oui	Oui	-
Système de vote côté client et serveur	Oui	Oui	-
Système de favoris/playlist	Oui	Oui	-
Nettoyage de la base de données côté serveur et client	Oui	Oui	-
Voir la liste des serveurs accessibles côté client	Oui	Oui	-
Accéder au serveur	Oui	Oui	-
Un client ne peut pas enregistrer deux fois la même chanson durant le même événement	Oui	Oui	-
Un client doit pouvoir supprimer une chanson de ses favoris ou ses playlists	Oui	Partiellement	-
Un client doit pouvoir supprimer une playlist toutes les chansons contenues dans ladite playlist	Oui	Partiellement	-
Support d'autres formats de musique	Non	Oui	Ajout du support du WAV.
Taille de fenêtre non-fixe	Non	Oui	-
Fusionner le code de l'application serveur et client	Non	Oui	Choix au démarrage
Filtres de recherche	Non	Non	-
Intégration de services externes	Non	Non	-
Système de transition dynamique entre chansons	Non	Non	-
Ajout d'une dimension communautaire	Non	Non	-
Définir des utilisateurs du système comme administrateurs	Non	Non	-
Configuration avancée du serveur	Non	Non	-

Fig. 19: Fonctionnalités implémentées

10 Améliorations envisagées

- Revoir l'architecture du projet pour mieux séparer les entités, avec le patron Observable-Observeur, par exemple, ce qui permettrait de notifier, à qui veut entendre, des informations.
- Rendre tous les messages et commandes asynchrones afin de minimiser les ressources et ne pas bloquer toute l'application lorsqu'il y a beaucoup de charge.
- Se passer des Singleton afin de rendre notre code plus indépendant.
- Mieux gérer la concurrence.
- Réaliser toutes les fonctionnalités optionnelles envisagées dans le cahier des charges.

11 Difficultés et problèmes rencontrés

Les points suivants ne sont pas forcément des difficultés mais surtout des points sur lesquels nous avons passé beaucoup plus de temps que prévu.

- Base de données : la gestion de la base de données à l'aide de Hibernate nous a pris énormément de temps à certains moments car il a fallu bien comprendre le fonctionnement de ce dernier et comment il évalue les objets afin de savoir s'ils doivent être sauvegarés ou non dans la base de données.
- Interfaces réseaux : le Multicast demandant d'utiliser une interface réseau précise, nous avons été confrontés à un problème sur certaines machines car il y avait plus d'une interface réseau de disponible. Avec de la chance, la première sélectionnée par Java était la bonne, mais pour certaines machines, avec des interfaces réseaux virtuelles, cela à posé problème car le Multicast était émis ou reçu sur la mauvaise interface.
- Interface graphique : l'interace graphique n'a pas posé de problème en soi mais l'implémentation a pris plus de temps que prévu pour rendre le programme davantage ergonomique et beau.

12 Conclusion

En conclusion, nous avons essayé de réaliser un programme qui regroupe les qualités suivantes :

- Code propre, facile à comprendre et réutilisable.
- Documentation claire et exhaustive du code.
- Facilité à utiliser et à comprendre pour des utilisateurs néophytes.
- Niveau d'abstraction le plus élevé possible.

Nous pensons avoir atteint ces objectifs. Il y a encore des points à améliorer mais nous avons réussi à produire un prototype fonctionnel qui réponde à la quasi totalité des points du cahier des charges.

13 Bilans personnels

13.1 Ludovic

J'ai la fierté de pouvoir me dire que ce projet de semestre s'est très bien passé. J'ai l'impression que l'on a toujours su communiquer dans le respect et en tenant compte des points de vue de chacun à la construction du projet. Cela a permis de pouvoir créer une réelle cohésion de groupe afin de réaliser quelque chose, qui n'était à la base qu'une idée sur papier, de fonctionnel et qui correspond quasiment à la version à laquelle nous avons réfléchi en tout début de projet.

La charge de travail était évidemment conséquente, mais la qualité de travail réalisée par mes collègues ainsi que la volonté de vouloir faire bien et mieux a permis de pouvoir réaliser à la fois quelque chose de beau visuellement mais aussi beau au niveau de la programmation.

Ce projet m'a permis de pouvoir approfondir mes connaissances et compétences techniques ainsi que mes compétences de chef de projet.

On se rend vite compte que la gestion d'une équipe n'est pas une chose aisée, mais qu'avec de bons collègues et les bons outils, même si tout ne se passe pas comme prévu, on arrive à atteindre les objectifs visés.

Mon seul regret est de ne pas avoir pu mieux impliquer tout le monde sur le développement.

13.2 Lucas

C'est la première fois que je travaille sur un projet d'une telle envergure et avec autant de personnes impliquées dans le développement.

Je suis très content de la cohésion du groupe et même si certaines personnes ont moins participé au niveau du code, elles ont su apporter leur expérience quant aux réflexions sur ce dernier. Nous avons toujours avancé dans la bonne entente et chacun a su être à l'écoute des critiques constructives des autres. Même si nous avons décidé de partager le travail en trois groupes, chacun a participé au développement de chaque partie de l'application.

Pour ce qui est du management du groupe, Ludovic a su donner des directives très claires pour que le projet avance. Mis à part cela, je n'ai pas vraiment ressenti de hiérarchie dans le groupe et c'était vraiment plaisant. Chacun avait son mot à dire et personne n'était mis à l'écart. Tout le monde a su être à l'écoute de chacun.

D'un point de vue personnel, je suis très heureux du résultat final. Les objectifs principaux que nous nous étions fixés ont quasiment tous été complètement remplis. Ce projet m'a permis de découvrir différentes technologies comme JavaFX ou Hibernate que je pourrais réutiliser dans certains de mes projets personnels.

En conclusion, ce projet m'a donné envie d'en réaliser d'autres de la même ampleur et avec le même type de personnes. La charge de travail était certes conséquente mais ce n'en fut pas moins un plaisir d'arriver à ce résultat.

13.3 Denise

En tant que première expérience dans un projet qui part de zéro et qui finit sur un programme fonctionnel, je peux dire que j'ai appris énormément, que ce soit au niveau technique, aussi bien qu'au niveau relationnel.

Du point de vue de la gestion du projet, j'ai le sentiment que Ludovic Delafontaine nous a permis à tous de rester sereins du début à la fin. En effet, il a eu la capacité de se remettre en question tout au long du projet, de nous permettre de lui dire si quelque chose n'allait pas bien et de traiter chaque étape du projet avec énormément de tranquillité et d'assurance. Les membres du groupes ayant déjà participé à des projets auparavant ont également permis de rendre l'expérience plus rassurante et pédagogique. Je pense surtout que le fait que nous nous soyons vus régulièrement a permis une dynamique de groupe excellente.

Le seul regret que j'aie pu avoir durant ce projet est très certainement le fait que, malgré mon implication importante dans l'interface graphique au début celui-ci, je n'aie pas pu fournir autant de code que ce que j'aurais désiré, mon manque d'expérience en étant sûrement la raison. Je vois toutefois cela d'un côté positif : j'ai pu apprendre de ce que les autres ont fait en restant au courant de l'évolution du programme et en participant aux discussions qui ont permis de le faire évoluer et devenir ce qu'il est aujourd'hui.

Globalement, je pense que c'est une expérience qui restera gravée en moi et qui m'aura permis de bâtir d'excellentes bases en vue de mon futur dans les projets d'informatique.

13.4 David

Ce projet fut une expérience enrichissante sur plusieurs points :

- Le fait de devoir trouver une idée d'application et de devoir en rédiger le cahier de charges, de la développer de A à Z et de devoir en produire la documentation complète.
- Exercer le travail en équipe et ce que cela implique.
- Fixer un planning et se rendre compte que certaines parties avaient été mal estimées.

Je pense que notre équipe a bien fonctionné et que les tâches ont été réparties correctement, les personnes qui ont moins codé ayant fait plus de documentation. J'ai particulièrement apprécié l'engagement de Ludovic Delafontaine en tant que chef de projet qui a su synthétiser les problèmes pour nous et les communiquer lorsque cela était nécessaire. L'équipe avait une bonne cohésion et les échanges réguliers ont permis de bien faire évoluer le projet en même temps que de mettre à jour tout le monde sur ce qui avait été fait par les différents membres.

13.5 Thibaut

Ce projet a été une très belle expérience pour les raisons suivantes :

- Travailler dans un groupe où les membres ont des profils différents, car dans le groupe il y avait des membres avec des profils conceptuels, certains avec des profils organisationels et d'autres avec des profils pratiques. De mon côté, j'aimerais aussi dans le futur devélopper ce profil de conception qui est primordial pour un ingénieur.
- Réflechir sur une idée de projet et la réaliser.
- Planifier le projet dans l'intervalle de temps mis à disposition.

Pour la gestion du projet, le chef de groupe, Ludovic, a parfaitement bien joué son rôle, il a su optimiser la répartition des tâches en vue d'arriver à la solution obtenue et la motivation qu'il apportait à chaque membre du groupe de travail était remarquable, vu que les tâches étaient reparties en petits groupes de deux personnes. Son sens de l'écoute, sa capacité à communiquer et à déléguer a permis la bonne réalisation du projet. Ce fut un plaisir de travailler avec ce groupe. Mes collègues ont toujours été disponibles pour répondre à mes questions et m'éclairer sur certains points. J'ai été aussi un peu fustré par la différence de niveau de programmation et de conception entre les autres menbres du groupe et moi, je pense que je dois encore bien travailler mon côté conceptuel.

13.6 Yosra

N'étant pas une personne d'expérience dans le domaine puisque je n'ai pas eu l'occasion de participer à plusieurs projets informatiques auparavant, cette nouvelle expérience n'a pas été immédiate et facile.

Ensemble, nous sommes partis d'une simple idée à développer part la suite. Notre équipe a dû donc unir ses forces afin que le projet aboutisse et qu'on puisse atteindre les objectifs fixés au tout départ.
Certains membres du groupe, quant à eux, avaient déjà acquis assez d'expérience durant plusieurs projets informatiques auxquels ils ont déjà pu participer et étaient donc toujours à l'écoute et prêts à donner un coup de main en cas de besoin.

Pour ma part j'étais dans le groupe s'occupant plus de la partie liée au réseau. Cette expérience m'a donc appris, d'une part, à avoir une certaine flexibilité d'adaptation dans les différents domaines concernant un même projet. D'autre part, elle ma permis d'acquérir de grandes compétences tout comme l'aisance relationnelle, la prise d'initiative ainsi que qu'une bonne organisation.

Pour ce qui concerne la gestion du groupe, Ludovic Delafontaine a réussi à guider le projet et assurer son avancement. Nous étions tous impliqués dans la réalisation du projet et il était toujours à l'écoute.

Pour conclure, je dirai que cette expérience unique en son genre m'a bien marquée et a sans doute été un apprentissage très précieux qui me sera utile dans le monde professionnel.

14 Sources

- Capsule (capsule.io) - Site officiel de la librairie Capsule
- Gson (github.com/google/gson) - Site officiel de la librairie Gson
- Hibernate (hibernate.org) - Site officiel de la librairie Hibernate
- JaudioTagger (jthink.net/jaudiotagger) - Site officiel de la librairie JaudioTagger
- Java 8 API Specification (docs.oracle.com/javase/8/docs/api) - Site officiel de la documentation pour Java 8
- Stack Overflow (stackoverflow.com) - Site d'entre-aide pour la programmation

15 Annexes

15.1 Défintition des normes de programmation

15.1.1 Documentation

- La documentation des méthodes et du code sera en anglais
- La tabulation est de quatre espaces
- La longeur maximale des lignes de code est sur 120 caractères
- La notation doxygen (doxygen.org) sera utilisée pour documenter le code, avec le caractère `@` et non `\`
- Un en-tête de fichier (doxygen) sera utilisé pour chaque fichier
- Utilisation de la notation camel case pour les variables et méthodes
- Les variables de type statique sont en majuscules
- Ne pas utiliser de caractères spéciaux dans les variables et fonctions

15.1.2 En-têtes

Les en-têtes de fonctions et fichiers devront respecter la forme suivante :

```
1 /**
2  * This is a correct comment
3  */
```

```
1 /**
2  * This is also
3  * a correct comment
4  */
```

```
1 /**
2  * As well as this one in case that the documentation is very long and have
       "multiple parts".
3  *
4  * So we leave a bit of space for a better reading experience.
5  */
```

Ne pas écrire d'en-têtes commme ceci :

```
1 /** This comment starts at the very top...
2  * ...but should has started here.
3  */
```

```
1 /**
2  *
3  * Why do we leave empty spaces before and after ? Isn't it useless ?
4  *
5  */
```

15.1.3 Fonctions

L'en-tête des fonctions devra respecter la forme suivante :

```
/**
 * Description about the method
 *
 * @param x Short description
 * @param y Short description
 *
 * @return int The multiplication of x and y
 */
public int myMethod(int x, int y) {
    return x * y;
}
```

Au sein d'une fonction, ne pas utiliser la forme de commentaire multi-lines :

```
/**
 * This is a comment
 * inside a function
 */
```

car si l'on souhaite commenter une partie de la fonction, les caractères `*/` couperont la fonction en commentaire.

Utiliser la forme suivante :

```
//! This is a multi-line comment
//! inside a function
```

Les parenthèses des fonctions restent collées au nom de la fonction, comme en mathématiques :

```
public int iAmACoolFunctionWithoutAnySpaces(int iHateSpaces) {
    ...
}
```

```
...

int a = iAmACoolFunctionWithoutAnySpaces(4);

...
```

15.1.4 Variables

Pour commenter des variables, on utilise la forme :

```
//! The current volume of the application
private unsigned int volume;

//! The playlist containing all the songs
private ArrayList<Music> playlist;
```

15.1.5 En-tête des fichiers

L'en-tête des fichiers devra respecter la forme suivante :

```
/**
 * This class is an example.
 *
 * Its description continues on multiple lines without any problem.
    However, if a comment is too long (max. 120
 * characters), feel free to break the comment in two lines.
 */
```

15.1.6 Les instructions de branchement

Les instructions de branchement sont écrites de la façon suivante, le '_' illustrant un espace :

```
if_(condition1_&&_condition2)_{
    ...
}_else_{
    ...
}
```

```
while_(condition1_&&_condition2)_{
    ...
}
```

```
switch_(condition1_&&_condition2)_{

    case_1:
        ...
        break;
    case_2:
        ...
        break;
    default:
        ...
        break;
}
```

```
for_(int_i_=_0;_i_<_10;_++i)_{
    ...
}
```

15.1.7 Classes

```
class MaClasse {
    attributs
        - static
        - public
        - protected
        - private
    méthodes
        - static
        - public
        - protected
        - private
}
```

15.1.8 Exemple complet

```java
/**
 * This class is an example.
 *
 * Its description continues on multiple lines without any problem.
    However, if a comment seems too long (max. 120
 * characters), feel free to break the comment in two lines.
 */
class MaClasse {

    //! Comment before the member
    private final static int I_AM_A_STATIC_VARIABLE = 10;

    //! Comment before the member
    public String iAmAString;

    //! Comment before the member
    protected char iAmACharacter;

    //! Comment before the member
    private booleand amIABoolean;

    /**
     *  Description about the method
     *
     * @param x Short description
     * @param y Short description
     *
     * @return int The multiplication of x and y
     */
    public int myMethod(int x, int y) {

        //! This part is so important that it needs
        //! multiple lines to describe it
        return x * y;
    }

    /**
     *  Description about the method
     *
     * @param c Change the message
     *
     * @return void
     */
    public void myMethod2(boolean c) {

        //! Change the message
        if (c) {
            System.out.println("Hi");
        } else {
            System.out.println("Good bye");
        }
    }
}
```

15.2 Planification initiale et son évolution

15.2.1 Planification initiale retravaillée

Notre diagramme de Gantt n'étant pas adapté à ce genre de projet, il a dû être retravaillé selon un template fourni par M. Rentsch.

La version initiale non retravaillée se trouve au chapitre 15.2.3.

Répartition des heures par développeur et par semaine

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	Total [h]
Denise Gemesio	4	4	5	7	7	7	7	7	7	7	7	7	7	3	3	1	**90**
Ludovic Delafontaine	5	5	5	7	7	7	7	7	7	7	7	7	7	2	2	1	**90**
Lucas Elisei	4	4	5	7	7	7	7	7	7	7	7	7	7	3	2	2	**90**
David Truan	5	5	5	5	6	6	6	6	7	7	7	7	7	5	3	3	**90**
Thibaut Togue	5	5	5	5	6	6	7	7	7	7	7	7	7	3	3	3	**90**
Yosra Harbaoui	4	4	6	6.5	6	7	6.5	6	6	7	6	7	7	4	4	3	**90**
Total [h]	**27**	**27**	**31**	**37.5**	**39**	**40**	**40.5**	**40**	**41**	**42**	**41**	**42**	**42**	**20**	**17**	**13**	**540**

Fig. 20: Planification initiale retravaillée

15.2.2 Heures effectives

Voici les heures effectives que nous avons reportées depuis le journal de travail.

Répartition des heures par développeur et par semaine

	S1	S2	S3	S4	S5	S6	S7	S8	Pâques	S9	S10	S11	S12	S13	S14	S15	S16	Total [h]
Denise Gemesio	4	4	5	2	12	5.5	3	5.5	0	3.5	1.5	3	4.5	31.5	7.5	1.5	1.5	**95.5**
Ludovic Delafontaine	5	5	5	5.5	11.6	11.5	4.3	4.5	5.5	2.5	14.5	3.25	7.75	49	7.5	1.5	1.5	**145.4**
Lucas Elisei	4	4	5	2	6	8.5	1.5	3.5	8	2.5	8.25	6	8	36.25	3	1.5	1.5	**109.5**
David Truan	5	5	5	2	11	7.5	4.7	3.5	0	6.5	8.2	6	9	15.75	5.5	1.5	1.5	**97.65**
Thibaut Togue	5	5	5	2	8.5	7	7.5	6.5	1.5	4.5	1.5	3.5	3.5	26.5	5.5	1.5	1.5	**96**
Yosra Harbaoui	4	4	6	2	11	9.5	7	4.5	0	2.5	3.5	6	3.5	26.75	3	1.5	1.5	**96.25**
Total [h]	**27**	**27**	**31**	**15.5**	**60.1**	**49.5**	**28**	**28**	**15**	**22**	**37.45**	**27.75**	**36.25**	**185.75**	**32**	**9**	**9**	**640.3**

Fig. 21: Heures effectives

15.2.3 Diagramme de Gantt initial personnel

Nous mettons à disposition la première version de notre planification qui était basée sur un diagramme de Gantt.

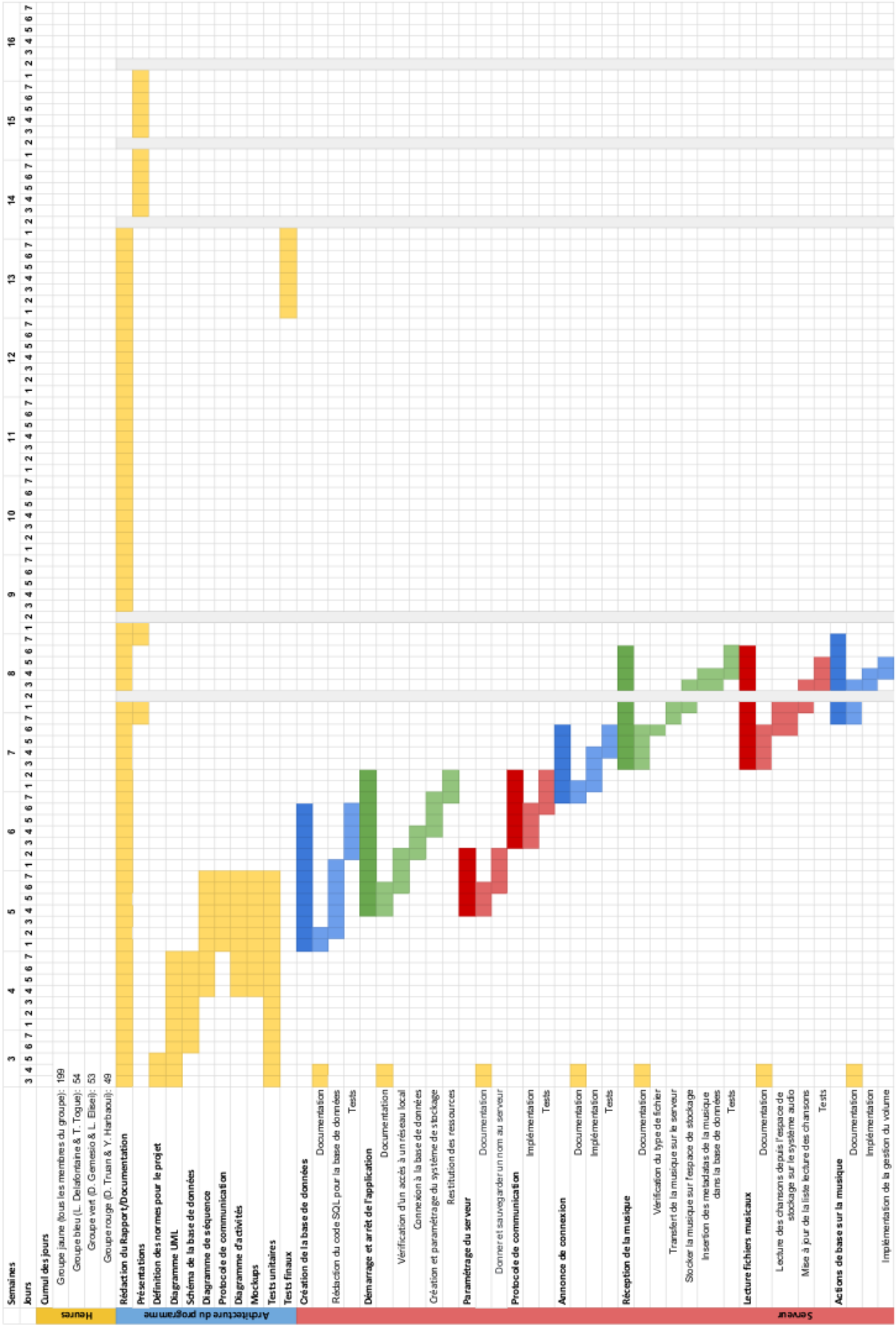

Fig. 22: Gantt initial personnel - partie 1

Fig. 23: Gantt initial personnel - partie 2

15.3 Cahier des charges

Le cahier des charges ci-dessous est une copie conforme de celui fourni au début du semestre.

15.3.1 Contexte et problématique

La musique a une part importante dans toute manifestation (anniversaire, concert, bar, soirée entre amis, etc.). Néanmoins, la musique est souvent gérée par une personne sur un appareil et il devient difficile pour une autre personne de changer la musique ou proposer la sienne.

15.3.2 Objectif

Nous souhaitons proposer une application de type client-serveur qui permet aux différents utilisateurs de proposer leur propre musique au serveur, qui les jouera sur un système audio au fur et à mesure de l'événement. Elle laissera aussi les utilisateurs gérer la file de lecture grâce à un système de votes.

15.3.3 Public cible

Notre application est adressée à des utilisateurs néophytes, de toutes générations. De ce fait, son utilisation doit être simple mais proposera des options qui raviront les utilisateurs souhaitant plus de contrôle sur le fonctionnement de l'application.

15.3.4 Limitations

Il s'agit ici de développer une application de type client-serveur multi-utilisateur avec interface graphique qui fonctionnera au niveau du réseau local. Il ne s'agit pas de réaliser une application client-serveur qui permettra de proposer de la musique à n'importe quel serveur n'importe où dans le monde. De plus, le serveur ne pourra pas gérer un nombre illimité d'utilisateurs et lira un nombre restreint de formats. L'application ne gérera pas la sécurité au niveau de la communication réseau tel que le spoofing de clients.

15.3.5 Fonctionnalités importantes

Les fonctionnalités listées ci-dessous, dans l'ordre d'importance, sont nécessaires au bon fonctionnement de l'application.

15.3.5.1 Commun aux deux parties de l'application

- **Fonction :** Démarrage et arrêt corrects du programme
  - Objectif : Préparer les ressources et les nettoyer correctement
  - Description : -
  - Contraintes : -
- **Fonction :** Droits client-serveur
  - Objectif : Ne pas autoriser des actions interdites de la part des clients au niveau du serveur et inversément.
  - Description :
    - Les clients ne peuvent pas modifier les paramètres du serveur
    - Les serveurs ne peuvent pas modifier les clients
  - Contraintes : -
- **Fonction :** Notification des actions
  - Objectif : Ne pas laisser les utilisateurs dans le doute de la bonne réalisation ou non des actions qu'ils effectuent
  - Description :
    - Les résultats de toutes les actions effectuées par les utilisateurs doivent être notifiées
    - Les erreurs ou les limitations doivent être notifiées
  - Contraintes : -

15.3.5.2 Côté serveur

- **Fonction :** Paramétrages basiques du serveur
  - Objectif : Donner un nom au serveur
  - Description : Donner la possibilité aux clients de savoir sur quel serveur ils vont se connecter.
  - Contraintes : -
- **Fonction :** Effectuer une annonce de connexion
  - Objectif : Autoriser les clients à se connecter au serveur
  - Description : Effectue une annonce dans le réseau local pour avertir les clients que le serveur est disponible à recevoir de la musique
  - Contraintes : Se limite au réseau local
- **Fonction :** Réception de la musique
  - Objectif : Ajouter de la musique sur le serveur
  - Description : Le client transfère une chanson au serveur qui sera ensuite lue. Le client est notifié du résultat.
  - Contraintes :
    - Une chanson ne peut être envoyée qu'une fois sur le serveur
    - + voir point "Accepter ou refuser l'ajout de nouvelles chansons"
- **Fonction :** Lecture des fichiers MP3 et M4A
  - Objectif : Le serveur supporte uniquement les fichiers MP3 et M4A
  - Description : -
  - Contraintes : Seuls fichiers supportés au début
- **Fonction :** Ajout de la musique à la base de données/système de stockage
  - Objectif : Sauvegarder les chansons pour la liaison entre l'application et le système de stockage
  - Description : Les metadatas (si présentes) ainsi que les données suivantes sont enregistrées dans la base de données :
    - Artiste
    - Titre de la chanson
    - Album
    - Durée de la piste
    - Chemin du fichier
    - Date d'ajout de la piste
    - Date de lecture de la piste
  - Contraintes : Voir le point "Accepter ou refuser l'ajout de nouvelles chansons"
- **Fonction :** Actions de base sur la musique
  - Objectif : Effectuer des actions sur la lecture de la musique
  - Description : Passer à la chanson suivante, remettre une des chansons précédentes, mettre sur pause, arrêter
  - Contraintes : -
- **Fonction :** Interface utilisateur
  - Objectif : Interface simplifée pour utiliser le logiciel
  - Description : Voir le mockup en annexe
    - Ajout de chansons locales à l'application
    - Récupération des metadatas des chansons et mise en liste de lecture
    - Voir les chansons en cours de lecture (queue de lecture)
  - Contraintes : La fenêtre ne peut pas être redimensionnée
- **Fonction :** Contrôle du volume de la musique
  - Objectif : Proposer, dans l'application, le réglage du volume
  - Description : -
  - Contraintes : -
- **Fonction :** Accepter ou refuser l'ajout de nouvelles chansons
  - Objectif : Accepter ou refuser aux clients de mettre de la musique sur le serveur
  - Description : Si le client tente de mettre de la musique alors que les options suivantes sont activées, l'action du client est refusée :
    - Nombre limité de transferts en parallèle entre tous les clients
    - Nombre limité de transferts de chansons par client
    - Limitation du nombre de chansons que le serveur autorise à avoir dans la queue de lecture
    - Limitation selon l'espace disque
    - Limitation selon la taille du fichier envoyé
    - Limitation selon le type de fichier envoyé
  - Contraintes : -

- **Fonction :** Système de vote
  - Objectif : Permettre aux utilisateurs de changer de chanson ou d'en prioriser une
  - Description :
    - Si un certain poucentage d'utilisateurs souhaite changer de chanson, le programme passe à la suivante
    - Une chanson qui a un vote positif remonte dans la queue de lecture
    - Une chanson qui atteint un nombre trop élévé de votes négatifs est supprimée de la playlist, de la base de données et du système de stockage
  - Contraintes : Un client ne peut voter qu'une fois pour une chanson
- **Fonction :** Système de favoris/playlist
  - Objectif : Permettre au client de sauvegarder les metadatas des chansons qui lui plaisent dans la base de données locale de ce dernier
  - Description :
    - Une playlist par défaut est créée automatiquement lors de la configuration du serveur et toutes les chansons y sont enregistrées.
    - Si un client souhaite enregistrer une chanson, le serveur lui envoie les metadatas de cette chanson et ces dernières sont enregistrées dans les favoris ou dans une playlist du client. Possibilité de sauvegarder les éléments suivants :
      - toute la musique qui a été jouée durant l'événement
      - toute la musique qui a été jouée depuis que le client s'est connecté pour la première fois à l'événement
      - des chansons indépendantes
  - Contraintes : Un client ne peut pas enregistrer deux fois la même chanson durant le même événement
- **Fonction :** Nettoyage de la base de données
  - Objectif : Libérer de la place sur le serveur
  - Description : Permet de supprimer la musique qui correspond aux critères suivants, à choix :
    - n'existe plus sur le disque
    - n'a pas été lue durant l'événement
    - n'a pas été lue depuis une certaine date
  - Contraintes : Nettoyage automatique si la capacité de stockage est limitée

15.3.5.3 Côté client

- **Fonction :** Voir la liste des serveurs accessibles
  - Objectif : Permet de choisir sur quel serveur se connecter
  - Description : Lorsque l'application est lancée, le client voit les serveurs accessibles par leur nom et peut s'y connecter
  - Contraintes : -
- **Fonction :** Accéder au serveur
  - Objectif : Accéder aux fonctionnalités du serveur
  - Description : Voir la liste de lecture, voter pour changer de chanson ou réorganiser la queue de lecture
  - Contraintes : Les fonctionnalités sont limitées selon la configuration du serveur
- **Fonction :** Ajouter de la musique au serveur
  - Objectif : Permet au client d'ajouter de la musique sur le serveur pour la lecture
  - Description : Un client peut ajouter sa musique locale à la queue de lecture du serveur
  - Contraintes :
    - Le serveur peut refuser l'ajout de la musique si la configuration de ce dernier n'autorise plus l'ajout de nouvelles chansons
    - Le client ne supporte que les fichiers avec des extensions .mp3 et .m4a
- **Fonction :** Interface utilisateur
  - Objectif : Interface simplifée pour utiliser le logiciel
  - Description : Voir le mockup en annexe
    - Voir les serveurs accessibles
    - Pouvoir se connecter sur un serveur
    - Voir la queue de lecture du serveur
  - Contraintes : La fenêtre de peut pas être redimensionnée
- **Fonction :** Système de vote
  - Objectif : Changer de musique, agencer la queue de lecture
  - Description : Donne la possibilité aux utilisateurs de changer ou arranger la musique en fonction des goûts
  - Contraintes : Un client qui vote deux fois pour la même action voit sa deuxième action refusée

- **Fonction :** Système de favoris/playlist
  - Objectif : Sauvegarder les chansons pour les retrouver après l'événement
  - Description : Sauve la queue de lecture dans la base de données locale - identique à la base de données du serveur, sans le chemin d'accès du fichier - du client selon qu'il souhaite récupérer toute la musique jouée pendant la soirée uniquement dès qu'il s'est connecté pour la première fois ou qu'il souhaite récupérer des chansons indépendantes avec possibilité de créer des playlists
  - Contraintes :
    * Un client ne peut pas enregistrer deux fois la même chanson durant le même événement
    * Un client doit pouvoir supprimer une chanson de ses favoris ou ses playlists
    * Un client doit pouvoir supprimer une playlist avec toutes les chansons contenues dans ladite playlist

15.3.6 Fonctionnalités optionnelles

Les fonctionnalités listées ci-dessous ne sont pas nécessaires au bon fonctionnement de l'application mais pourront être réalisées si le temps le permet. Ces dernières ne sont pas dans un ordre précis.

15.3.6.1 Commun aux deux parties de l'application

- **Fonction :** Support d'autres formats de musique
  - Objectif : Etendre les possibilités de lecture du serveur
  - Description : FLAC, ALAC, etc.
  - Contraintes : -
- **Fonction :** Taille de fenêtre non-fixe
  - Objectif : Permet de pouvoir utiliser l'application sur n'importe quelle écran avec n'importe quelle résolution
  - Description : -
  - Contraintes : Taille minimum requise
- **Fonction :** Fusionner le code de l'application serveur et client
  - Objectif : Permet à n'importe quel client de devenir serveur et inversément
  - Description : -
  - Contraintes : Les deux interfaces graphiques se voudront très similaires, sans possibilité de modification des paramètres du serveur de la part des clients
- **Fonction :** Filtres de recherche
  - Objectif : Rechercher des chansons sur le serveur (queue de lecture et playlist)
  - Description : Rechercher et mettre dans les favoris ou voter pour une chanson en particulier
  - Contraintes : Recherche limitée aux informations contenues dans la base de données
- **Fonction :** Intégration de services externes
  - Objectif : Permettre la lecture de chansons issues de services externes (SoundCloud, YouTube, etc.)
  - Description : -
  - Contraintes : La sauvegarde de la session de l'utilisateur est encore à définir
- **Fonction :** Système de transition dynamique entre chansons
  - Objectif : Transition fluide entre les chansons et les genres
  - Description : Système de transition dynamique entre chansons selon le rythme ou le genre de la musique
  - Contraintes : -
- **Fonction :** Ajout d'une dimension communautaire
  - Objectif : Interaction entre les utilisateurs
  - Description :
    * Ajout de comptes utilisateurs
    * Partage des chansons et playlists
    * Un compte utilisateur possède une notion de karma et de "rewards" associé au karma
      · Un utilisateur voit son karma augmenter pour une bonne action (voter positivement pour une chanson, mettre sur le serveur de la musique)
      · Un utilisateur voit son karma diminuer pour une mauvase action (voter négativement pour une chanson)
      · Selon le karma, l'utilisateur peut recevoir des avantages
  - Contraintes : Base de données accessible depuis n'importe où pour tous les utilisateurs
- **Fonction :** Définir des utilisateurs du système comme administrateurs
  - Objectif : Autoriser certains utilisateurs à avoir plus de droits que les autres

  - Description : Leur permettre de mettre la musique sur pause et régler
  - Contraintes : -

15.3.6.2 Côté serveur

- **Fonction :** Configuration avancée du serveur
  - Objectif : Options poussées pour la configuration du serveur
  - Description :
    * Lecture séquentielle : les chansons sont lues les unes après les autres, en ne tenant pas compte des votes : premier arrivé, premier servi
    * Lecture démocratique : les chansons sont lues en fonction des préférences des utilisateurs grâce au système de vote
    * Lecture aléatoire : les chansons sont lues aléatoirement quels que soient les votes
    * Lecture hybride : alternation entre les chansons populaires, qui ont beaucoup de votes, et moins populaires, qui ont moins ou pas de votes
    * Chemin de stockage de la musique : où est enregistrée la musique
    * Une musique ne peut être relue que après un certain nombre de minutes après sa première lecture
  - Contraintes : Refuser les actions effectuées par les clients si elles ne respectent pas la configuration du serveur

15.3.7 Résumé et schémas de fonctionnement du programme

1. Un serveur est lancé et est configuré selon les préférences de la personne qui gère le serveur.
2. Le serveur est démarré et les clients peuvent s'y connecter.
3. Le client est lancé et voit la liste des serveurs disponibles.
4. Le client se connecte sur un serveur.
5. Une fois connecté, il peut effectuer les fonctionnalités paramétrées sur le serveur :
   - Proposer de nouvelles chansons.
   - Voter pour changer ou organiser la playlist.
   - Enregistrer en favoris des chansons ou la liste de lecture.
6. Le serveur enregistre la chanson en local et la lit au fur et à mesure de l'événement, en fonction des éventuelles préférences des utilisateurs.
7. Une fois l'événement terminé, la musique est conservée sur le serveur jusqu'à ce que l'administrateur décide de nettoyer la base de données ou que la capacité maximum de stockage soit atteinte.
8. Le client conserve une copie des métadonnées des chansons qui lui ont plu, dans sa base de données locale et peut, de ce fait, retrouver les morcaux qui lui ont plu lors de cet événement.

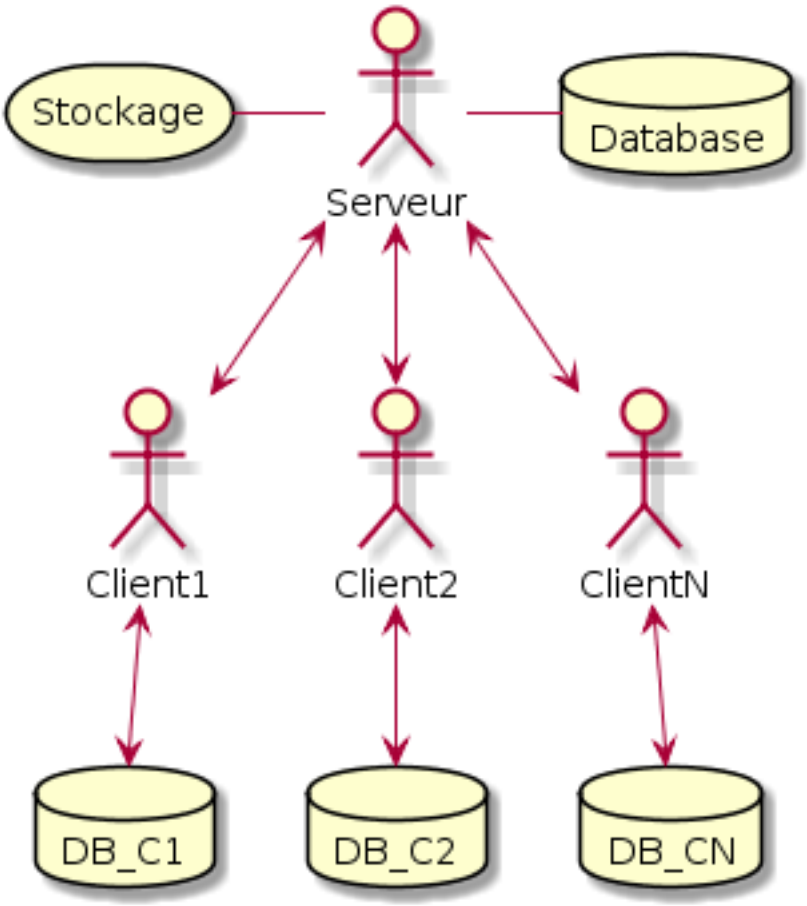

FIG. 24: Schéma préliminaire du fonctionnement général

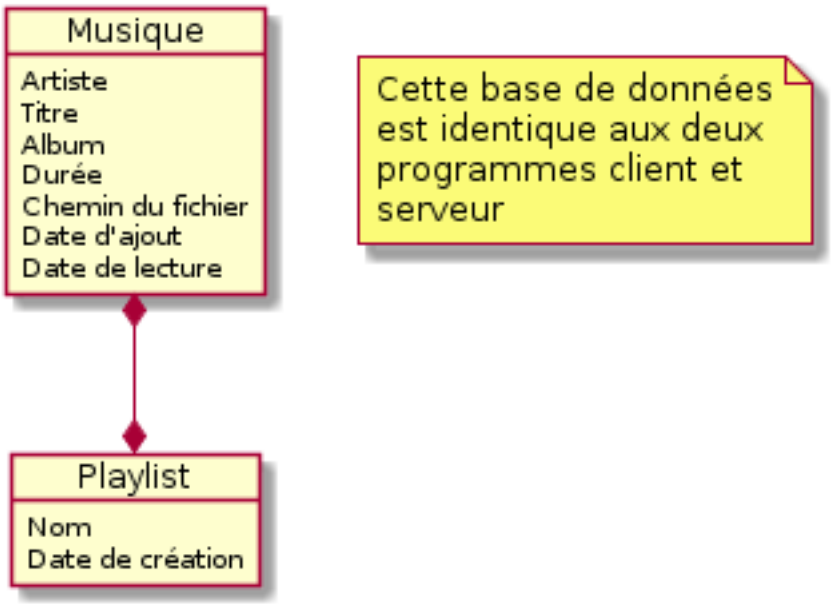

FIG. 25: Schéma préliminaire de la base de données

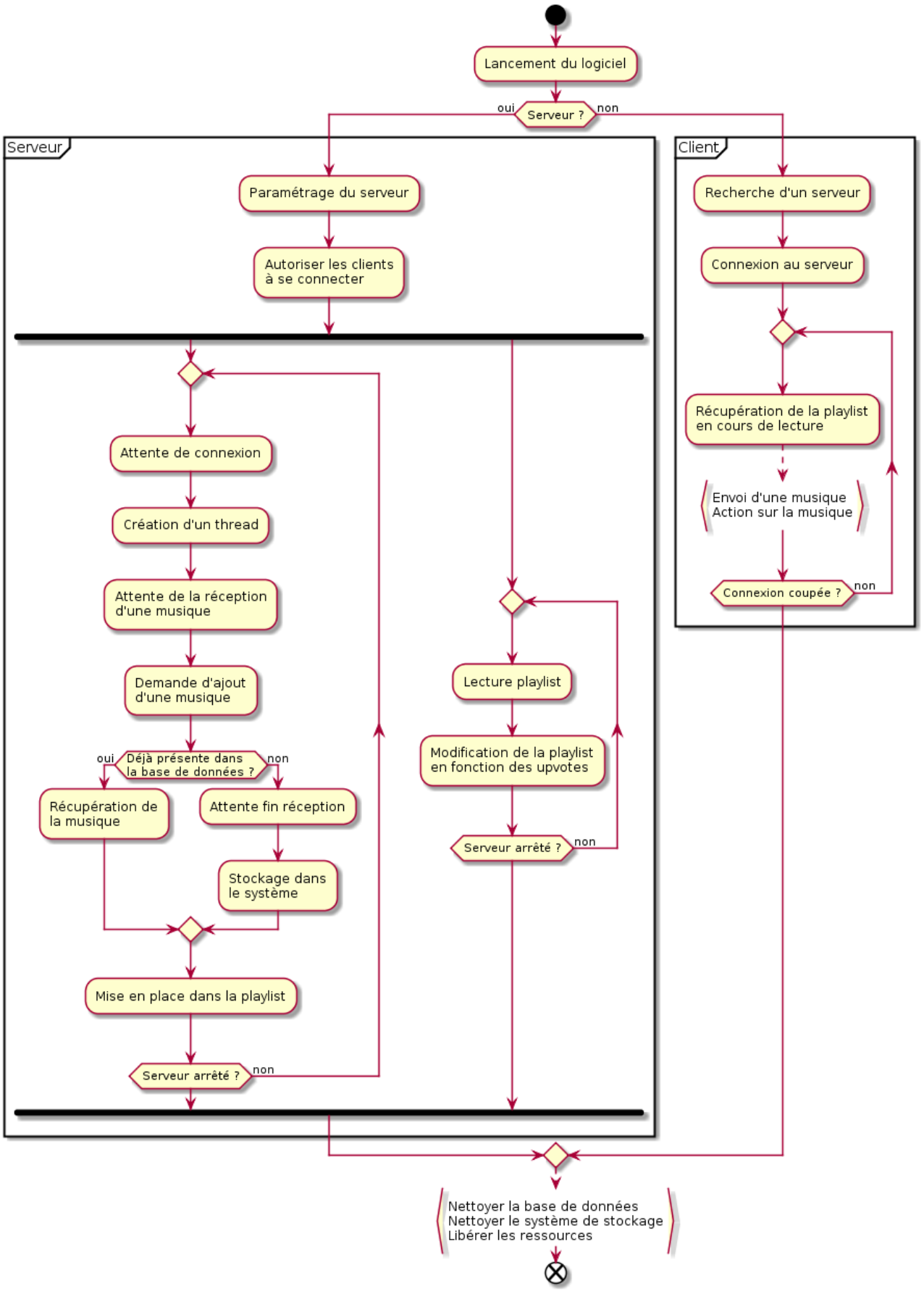

FIG. 26: Schéma préliminaire du schéma d'activité

15.3.8 Spécifications techniques

L'application sera réalisée à l'aide des technologies suivantes :

- Java (java.com) : pour la réalisation du programme
- JavaFX (docs.oracle.com/javafx) : pour la réalisation de l'interface graphique
- SQLite (sqlite.org) : pour la réalisation de la base de données
- JSON (json.org) : pour l'interaction entre le client et le serveur
- + différentes librairies qui pourraient être découvertes durant la conception du programme

Et s'appuiera sur les outils suivants pour sa réalisation :

- Git (git-scm.com) / GitHub (github.com) : pour la gestion de versions du projet
- Travis (travis-ci.org) : pour les tests unitaires afin de s'assurer du bon fonctionnement de l'application
- PlantUML (plantuml.com) : pour la génération des différents schémas (diagrammes de classes, diagrammes de séquences, diagrammes pour le schéma relationnel, etc.)
- Pencil (pencil.evolus.vn) : pour la création de mockups et interfaces simplifiées
- + différents outils qui pourraient être découverts durant la conception du programme

Et s'exécutera sur les systèmes d'exploitation suivants :

- Windows (version 10, au mimimum)
- Linux
- Mac OS

15.3.9 Ressources à disposition et organisation

Le projet se déroulera sur tout le semestre pour un total de 90 heures de travail par personne, soit 540 heures de travail effectif sur 14 semaines. Cela représente environ six heures de travail par personne par semaine. Une métodologie AGILE sera appliquée afin d'avoir un suivi de l'évolution du travail.

15.3.10 Rendu

En plus des points évoqués dans les contraintes du cours PRO et selon les fonctionnalités importantes, le rendu sera de la forme suivante :

- Un fichier .jar qui représentera le programme côté serveur
- Un fichier de configuration du serveur
- Un fichier .jar qui représente le programme côté client

15.3.11 Indicateurs et évaluation des résultats

Les différentes itérations de la méthodologie AGILE permettront de quantifier l'avancement du travail et sa bonne réalisation.

15.3.12 Difficultés envisagées

Les éléments suivants semblent être ceux qui devront prendre plus de temps pour leur réalisation au vu de leur complexité :

- Envoi et gestion de fichiers
- Interface graphique
- Lecture de fichiers musicaux

15.3.13 Annexes

- Planification
- Mockups de l'application

15.4 Journal de travail

Le journal de travail débute le 15.03.2017

15.4.1 En groupe

- 23.05.2017
  - Discussion autour du rapport et de sa rédaction (1h30)
- 16.05.2017
  - Discussion autour des derniers bogues à réparer (1h30)
- 09.05.2017
  - Discussion autour des derniers détails à régler (1h30)
- 02.05.2017
  - Mise au point de l'avancement dans le projet (1h30)
- 28.04.2017
  - Discussion autour de l'avancement de chacun (1h00)
- 25.04.2017
  - Mise au point des modifications faites durant les vacances (1h30)
- 11.04.2017
  - Présentation intermédiaire du projet + discussions diverses (1h30)
- 10.04.2017
  - Réalisation de la présentation (2h00)
- 04.04.2017
  - Discussion générale : organisation de l'interface graphique, revue des problèmes liés aux classes Track, Playlist et PlaylistManager, discussion autour de l'utilité de Player, discussion autour de NetworkManager, discussion autour de la présentation, création et rédaction de celle-ci (1h30)
- 31.03.2017
  - Discussion autour de la base de données et de l'avancement de chacun (2h00)
- 28.03.2017
  - Discussion autour de l'interface graphique et du FileManager (1h30)
- 24.03.2017
  - Discussion permettant de savoir où chacun en est et questions sur des doutes divers concernant des choix d'implémentation et architecture (3h00)
- 21.03.2017
  - Mise en place de Apache Maven et discussion sur le projet (1h30)
- 17.03.2017
  - Distribution des rôles, discussion autour de l'architecture et définition de celle-ci (2h00)

15.4.2 Ludovic Delafontaine

- 29.05.2017
  - Corrections et modifications du manuel utilisateur (2h00)
  - Elaboration des tests (depuis 12h30)
- 28.05.2017
  - Rédaction du rapport
  - Améliorations du template
  - Ajout d'images
  - Réalisation du manuel d'utilisateur avec Yosra
  - Total du travail : 11h00
- 27.05.2017
  - Créations de divers schémas pour le rapport
  - Rédaction et améliorations du rapport
  - Corrections dans ServerCore vis à vis de l'interaction avec la base de données
  - Total du travail : 10h00
- 26.05.2017
  - Corrections de bugs avec la base de données
  - Améliorations des logs
  - Améliorations du rapport
  - Corrections de problèmes aec les sessions
  - Total du travail : 12h00
- 25.05.2017
  - Ajout des commandes manquantes aux Cores
  - Documentation
  - Corrections de Null Pointers Exceptions
  - Total du travail : 12h00
- 24.05.2017
  - Docmentation de tout le code des classes (1h30)
- 23.05.2017
  - Améliorations de la base de données (2h00)
- 22.05.2017
  - Finalisation du protocol applicatif (3h00)
  - Documentation (1h00)
  - Fusion du travail de chacun (1h00)
  - Tests (1h00)
- 21.05.2017
  - Ajout de commandes au niveau des core pour gérer le volume et demander la track suivante (1h00)
  - Documentation (0h45)
- 20.05.2017
  - Documentation du code et mise au propre (1h30)
- 19.05.2017
  - Mise au propre du File Manager (3h00)
- 10.05.2017
  - Fusion du travail de tout le monde sur la branche master (0h45)
- 09.05.2017
  - Reflexion avec David Truan au niveau de la nouvelle implémentation client/serveur (1h00)
- 07.05.2017
  - Améliorations des notions de client/serveur (3h00)
- 06.05.2017
  - Améliorations des notions de client/serveur (6h00)
- 05.05.2017
  - Revue de code afin de pouvoir tout fusionner (2h00)
  - Améliorations des notions de client/serveur (2h00)
- 20.04.2017
  - Suite des tests et version fonctionnelle du player (1h30)
- 19.04.2017
  - Reprise du player, playlist manager et filemanager et intégration avec le reste du projet (4h00)
- 11.04.2017
  - Début de la mise à jour du schéma UML selon la réelle implémentation (1h00)
- 05.04.2017

  - Ajout de la classe Playlist et de sa table associée pour la base de données (01h00)
  - Ajout de la classe permettant de récupérer des propriétés depuis un fichier de configuration (00h30)
  - Documentation des différentes classes (00h30)
- 03.04.2017
  - Finalisation de la classe Track avec l'ajout de l'interface DatabaseObject (0h30)
  - Merge avec master pour Track (0h20)
- 31.03.2017
  - Suite de la base de données avec les classes Playlist, PlaylistTrack et PlaylistTrackId (2h30)
- 27.03.2017
  - Création de la classe Playlist et tests avec la base de données associée (5h00)
- 26.03.2017
  - Mise en place et configuration des outils de compilation (Maven) (2h00)
  - Début de la classe Track avec sa table dans la base de données à l'aide de Hibernate (1h30)
- 22.03.2017
  - Player (1h00)
  - Tests de lecture de fichiers audio (0h45)
  - Tests de récupération des metadatas des fichiers (0h20)
- 21.03.2017
  - Avancement dans la base de données (1h00)
  - Corrections et améliorations (0h30)
- 18.03.2017
  - Création du schéma de la base de données (0h30)
  - Mise à jour du diagramme UML (0h30)
- 15.03.2017
  - Ajout des éléments manquants dans les PV (1h00)
  - Création du journal de travail (0h15)
  - Corrections du diagramme de séquence (0h30)
  - Documentation sur Reflexion (0h45)

15.4.3 Lucas Elisei

- 29.05.2017
  - Elaboration des tests et impression du rapport (4h00)
- 28.05.2017
  - Rédaction du rapport (3h00).
- 27.05.2017
  - Correction d'un bogue qui ne mettait pas correctement à jour la liste des serveurs côté client (0h15).
  - Rédaction du rapport (1h00).
- 26.05.2017
  - La barre d'avancement du morceau en cours de lecture se met maintenant correctement à jour côté client (1h30).
  - Les morceaux déjà joués se mettent maintenant correctement à jour côté client (1h30).
  - Correction de bogues liés au rafraîchissement de l'interface graphique côté client (3h00).
  - Correction d'un bogue qui changeait l'ordre des morceaux déjà joués côté client (1h30).
  - Ajout du panneau des réglages à l'interface graphique (4h00).
  - Correction d'un bogue qui affichait mal le nom du serveur auquel le client est connecté (0h30).
  - Total du travail : 11h30
- 25.05.2017
  - Correction d'un bogue qui empêchait les votes des morceaux de se mettre à jour côté client (1h30).
  - Correction d'un bogue qui empêchait le serveur d'upvote des morceaux (0h30).
  - Les morceaux s'enregistrent correctement dans la base de données (0h30).
  - Correction d'exceptions levées par la base de données (1h00).
  - Changement de la logique de la playlist éphémère (2h00).
  - La vue du morceau en cours de lecture se met maintenant à jour côté client et serveur (2h00).
  - Correction d'un bogue qui empêchait des communications parallèles avec la base de données (0h30).
  - La vue du morceau précédent se met maintenant correctement à jour côté client (1h00).
  - La barre de volume se met maintenant correctement à jour côté client (1h00).
  - Le bouton play/pause se met maintenant correctement à jour côté client (0h30).
  - Total du travail : 9h30
- 24.05.2017
  - Correction de plusieurs bugs d'affichage liés à l'interface graphique (2h00).
- 23.05.2017
  - Modification de la sérialisation de la playlist éphémère (1h00).
  - Meilleure gestion de la sélection d'une playlist au niveau de l'interface graphique (1h00).
- 22.05.2017
  - Ajout d'une fenêtre au démarrage pour choisir si l'on veut être client ou serveur (0h30).
  - Ajout de la possibilité de se connecter à un serveur depuis l'interface graphique (1h00).
  - Ajout de la possibilité de transférer une musique depuis l'interface graphique (1h00).
  - Modification de la sérialisation d'une musique et de la playlist éphémère (1h30).
- 16.05.2017
  - Correction d'un bogue qui ne terminait pas correctement le player (0h30).
  - Correction d'un bogue qui ne laissait pas favoriser les morceaux de la playlist éphémère (1h15).
  - Correction d'un bogue qui laissait la possibilité de voter pour les morceaux d'une playlist sauvegardée (0h15).
- 15.05.2017
  - Revue complète de la logique du PlaylistManager (3h30).
  - Intégration des actions de favoris à l'interface graphique (1h00).
- 10.05.2017
  - Création automatique de la playlist "Favoris" dans la base de données si celle-ci n'existait pas (2h00).
  - Fin de l'embellissement du panneau des playlists (1h00).
- 09.05.2017
  - Modification du player afin que le prochain morceau soit jouée automatiquement (0h30).
  - Début de l'embellissement du panneau des playlists (1h00).
- 06.05.2017
  - Finalisation de la fusion du panneau "morceau précédent" (0h30).
  - Correction de quelques bogues liés aux précédentes itérations (1h30).
- 03.05.2017
  - Fusion du panneau "playlist en cours de lecture" avec le code (1h00).
  - Implémentation d'une structure de données pour le tri des morceaux des playlists (2h00).
  - Début de la fusion du panneau "morceau précédent" avec le code (0h15).

- 02.05.2017
  - Fusion du panneau “morceau en cours de lecture” avec le code (1h00).
- 01.05.2017
  - Documentation quant à la fusion du code et de l’interface graphique (0h30).
- 23.04.2017
  - Implémentation du player dans l’interface graphique (1h00)
  - Découpage de l’interface graphique en modules (2h00)
- 21.04.2017
  - Implémentation de chargement de playlists dans l’interface graphique (4h00)
- 17.04.2017
  - Mise à jour de l’interface graphique (panneau central) selon les choix retenus lors de la précédente réunion. (1h00)
- 29.03.2017 (1h00)
  - Ajout des actions d’*upvote* et *downvote* pour les morceaux (seulement graphique).
- 28.03.2017 (4h00)
  - Ajout du style des cellules représentant des morceaux dans la playlist en cours de lecture (panneau central).
  - Tests du player et suite de la base de données (2h00)
- 21.03.2017
  - Interface graphique : premier jet (1h30)
  - Mise en place des principaux composants graphiques
  - Affichage d’une liste de playlists

15.4.4 David Truan

- 29.05.2017
  - Elaboration des tests (4h00)
- 28.05.2017
  - Finition du rapport et création des tableaux (3h).
- 27.05.2017
  - Finition des package Core et Network du rapport (3h00)
  - Création de différents schémas pour le rapport (1h00)
  - Correction du code pour le changement de l'interface (0h30)
  - Tests sur le programme (0h30)
- 26.05.2017
  - Continuation du rapport sur la partie Network et Core (3h00)
  - Codage des réactions de l'UI chez le client et des fonctions de vote manquante du serveur (2h00)
  - Codage d'une fenêtre pour choisir le nom du serveur (0h45)
  - Tests sur le programme (1h00)
- 25.05.2017
  - Début de l'explication des cores dans le rapport (1h30).
- 24.05.2017
  - Correction de bugs liés é la majorités des votes et au lancement des morceaux (1h00).
- 21.05.2017
  - Ajout des commandes pour les upvote/downvote et leur gestion dans UserSession (1h30).
  - Documentation et ajout de messages d'erreur (1h00).
  - Meilleure notification des nouveaux clients au serveur (0h30).
- 16.05.2017
  - Implémentation des checks si un morceau est dans la base de donnée (0h30).
- 15.05.2017
  - Reflexion de l'implémentation des contrôles sur les fichiers (1h00)
  - Implémentation de check de checksum MD5 pour les fichiers transférés (1h00).
  - Cleaning des Cores clients et serveurs (2h00)
- 14.05.2017
  - Mise en place de la decouverte de serveurs (1h30)
  - Amélioration de la communiation et fix de bug des Cores (2h00)
- 09.05.2017
  - Reflexion avec Ludovic Delafontaine au niveau de la nouvelle implémentation client/serveur (1h00)
- 04.05.2017
  - Lecture des Metadatas des fichiers et tests de la libraire audio utilisée (2h00)
  - Implémentation de la lecture des signatures des fichiers audios (1h00)
- 02.05.2017
  - Finition de quelques bugs du serveur (1h00)
  - Délégation du transfert de fichier au FileManager(0h10)
  - Refonte de la classe Session (0h30)
- 01.05.2017
  - Mise au propre du network (2h00)
  - Déconnexion client/serveur
  - Singletons des threads
  - Remise en forme du client
- 30.04.2017
  - Transfert de fichier .mp3 et implémentation dans le projet (2h00)
  - Mise au propre des classes du package network et finition du protocole (1h00)
  - Mise en place de la mise à jour de la playlist (1h00)
- 08.04.2017
  - Ajout d'une méthode pour choisir sa bonne interface et modification du code pour prendre en compte cela. (2h00)
  - Ajout de fonctionnalités au programme de test. (0h10)
  - Mise au propre rapide des classes et création de a classe Protocol et de packages client/server. (1h00)
- 01.04.2017
  - Tests pour le Multicast. Toujours des problèmes (2h00)
- 30.03.2017
  - Réflexion sur l'implémentation et début de code pour la découverte de serveurs par les clients (2h00)
- 25.03.2017

  - Documentation et première implémentation (test) du multicast en Java. (2h00)
  - Réévaluation de l'intêret de NetPort en tant que classe. (0h30)
- 21.03.2017
  - Meilleur division client/serveur et base du protocole (1h00)
- 20.03.2017
  - Essais et documentation sur la partie client/serveur (3h00)

15.4.5 Denise Gemesio

- 29.05.2017
  - Corrections et modifications du manuel d'utilisateur (2h00)
  - Elaboration des tests (4h00)
- 28.05.2017
  - Planification finale, création (0h30)
  - Planification finale, modification (1h00)
  - Correction du rapport (3h30)
  - Planification finale, modification (0h30)
  - Correction du manuel d'utilisateur (1h00)
- 27.05.2017
  - Relecture complète du rapport et corrections orthographiques et grammaticales (3h00)
  - Rédaction du rapport (UI) (3h00)
- 26.05.2017
  - Rédaction du rapport (UI) (5h00)
  - Corrections orthographiques et grammaticales (2h00)
- 25.05.2017
  - Rédaction du rapport (paquets playlist, utils et UI) (6h00)
  - Mise au propre du journal de travail (1h00)
  - Corrections orthographiques et grammaticales du rapport (2h00)
- 24.05.2017
  - Rédaction du rapport (2h00)
- 23.05.2017
  - Revue de l'introduction et objectifs du rapport (1h00)
- 19.05.2017
  - Modification du rapport (1h00)
- 15.05.2017
  - Analyse de l'implémentation du programme. Premier résumé de l'utilité de chacune des classes (2h00)
- 14.05.2017
  - Relecture et analyse de l'implémentation (1h30)
- 27.04.2017
  - Analyse des modifications au niveau de l'interface graphique, compréhension de la nouvelle structure (1h00)
- 11.04.2017
  - Réglage de la barre de progression de la musique (1h00)
- 10.04.2017
  - Réalisation de la présentation (1h00)
- 07.04.2017
  - Rédaction de la présentation intermédiaire (1h30)
- 02.04.2017
  - Fenêtre rendue "redimensionnable" au niveau de la partie player de l'interface (2h00)
- 26.03.2017
  - Intergace graphique : création du .fxml avec Scene Builder (4h00)
- 22.03.2017
  - Interface graphique : tutoriel et documentation (2h30)
  - Installation de Scene Builder et configuration de Intellij (1h00)
  - Tutoriel sur : http ://code.makery.ch/library/javafx-8-tutorial/fr/

15.4.6 Thibaut Togue Kamga

- 29.05.2017
  - Elaboration des tests (4h)
  - Création du document et début des tests (1h30)
- 28.05.2017
  - Reflexion sur les scénarios des tests (04h00)
- 27.05.2017
  - Corrections du rapport après les suggestions de Denise Gemesio (2h00)
  - Relecture du rapport(2h30)
- 26.05.2017
  - Rapport : description des technologies utilisées (1h30)
  - Rapport : description du paquet Session (2h30)
  - Définition du diagramme d'activité de l'échange de fichiers entre le client et le serveur, ensuite repris par Ludovic Delafontaine (2h00)
- 25.05.2017
  - Rapport : description des paquets Database et File (2h00)
  - Réalisation du diagramme UML de chaque paquet avec Plantuml, ensuite repris par Ludovic Delafontaine (2h30)
  - Réalisation du diagramme UML de tout le projet, ensuite repris par Ludovic Delafontaine (2h00)
- 24.05.2017
  - Rapport : description du paquet media (2h00)
- 23.05.2017
  - Test de l'application et discussions concernant le rapport (2h00)
- 18.05.2017
  - Début de rapport (définir la structure du rapport, introduction, objectif, description package Configuration du rapport) (3h00)
- 11.05.2017
  - Lecture du code contrôleur pour la documentation (2h30)
- 26.04.2017
  - Test du player avec le playlistManager (2h30)
- 17.04.2017
  - Finalisation du player et filemanager, validé par Ludovic (2h00)
- 10.04.2017
  - Implémentation du player sans playlistManager et tests (4h00)
- 06.04.2017
  - Tutoriel playerMedia pour l'implémentation du player (2h00)
- 03.04.2017
  - Implémentation du player avec SourceDataline mais inutile car impossible de lire les mp3 (4h00)
- 30.03.2017
  - Tutoriel sur les file poperties de Java et implémentation du fichier de configuration, modification du file Manager d'après la discussion avec le groupe pour la réservation de la mémoire et tests (3h30)
- 27.03.2017
  - Implémentation du fileManager (3h00)
- 22.03.2017
  - Tutoriel sur https://www.jmdoudoux.fr/java/dej/chap-hibernate.htm pour la réalisation de la couche persistance de l'application avec *ORM* (3h00)
- 20.03.2017
  - Test du fichier SQLite à travers des conteneurs Docker et correction des bugs (3h30)

15.4.7 Yosra Harbaoui

- 29.05.2017
  - Elaboration des tests (1h30)
- 28.05.2017
  - Rédaction du manuel d'utilisation. (3h00)
  - Mise à jour du journal du travail. (0h45)
  - Re-lecture du rapport. (2h00)
- 27.05.2017
  - Tests finaux de l'application et vérification du bon fonctionnement. (1h30)
  - Rédaction du manuel d'utilisateur. (4h00)
  - Rédaction du rapport. (2h00)
- 26.05.2017
  - Tests de l'application et vérification du bon fonctionnement. (1h30)
  - Rédaction du rapport. (2h30)
- 25.05.2017
  - Lecture du code et comprendre les parties ambigües. (2h30)
- 24.05.2017
  - Tests du bon fonctionnement de l'application. (1h00)
  - Corrections orthographiques du rapport. (1h00)
- 23.05.2017
  - Rédaction du rapport. (1h30)
  - Lecture des autres différentes parties implémentées. (2h00)
- 17.05.2017
  - Tests de l'implémentation des contrôles sur les fichiers (1h00)
- 15.05.2017
  - Documentation sur les contrôles sur les fichiers (1h00).
- 13.05.2017
  - Modification de la classe Session (1h30)
- 12.05.2017
  - Implémentation de la classe Session (2h00)
  - Tests (1h00)
- 01.05.2017
  - Transfert de fichier (1h00)
  - Tests (1h00)
- 10.04.2017
  - Réalisation de la présentation (1h00)
- 09.04.2017
  - Rédaction de la présentation intermédiaire. (1h00)
  - Analyse de l'implémentation du protocole de communication. (1h30)
  - Tests de choix de la bonne interface (1h00)
- 08.04.2017
  - Documentation pour le choix de la bonne interface (2h00)
- 31.03.2017
  - Implémentation simple d'une connexion client/serveur (3h00)
  - Tests de la connectivité (1h00)
- 27.03.2017
  - Documentation sur la configuration des outils de compilation (Maven) (1h00)
  - Documentation sur Hibernate (1h00)
- 25.03.2017
  - Documentation sur les différents "types" de communications entre un serveur et un client. (3h00)
- 21.03.2017
  - Documentation sur l'implémentation client/serveur. (2h30)
  - Installation de Scene Builder et configuration de Intellij. (1h00)