Application final

Applications distribuées

RODRÍGUEZ, RICARDO MARTÍN

Année scolaire 2016-2017
Master Informatique M1
ILSEN
Université d'Avignon et des Pays de Vaucluse

TABLE DES MATIÈRES

1 INTRODUCTION

Il fallait développer une application distribuée qui permette à des clients Android jouer des fichiers de musique. Les chansons se trouvent dans un ensemble de serveurs distants.

Le système comporte plusieurs modules :

- Application cliente Android
- Un webservice qui reçoit les commandes utilisateur.
- Un serveur central (Manager) qui orchestre l'accès à tous les modules
- Un module de parsing de commandes
- Un module de traitement de commandes
- Un méta-serveur qui connaît la liste globale de chansons
- Un ou plusieurs serveurs de fichiers musicaux (miniserveurs)

La technologie utilisée pour la communication est essentiellement Ice by ZeroC. Le module Manager a une interface Ice pour communiquer avec le webservice et les échanges entre le Métaserveur et les miniserveurs est faite vie IceStorm et IceBox.

Le webservice est un service REST développé en Python avec la bibliothèque web.py.

Modules non implémentés dans cette version :
À différence de l'idée initiale, présentée dans le Rapport initial, le système développé ne comporte pas un module de Reconnaissance de la parole. Plus bas j'explique les raisons de cette différence.

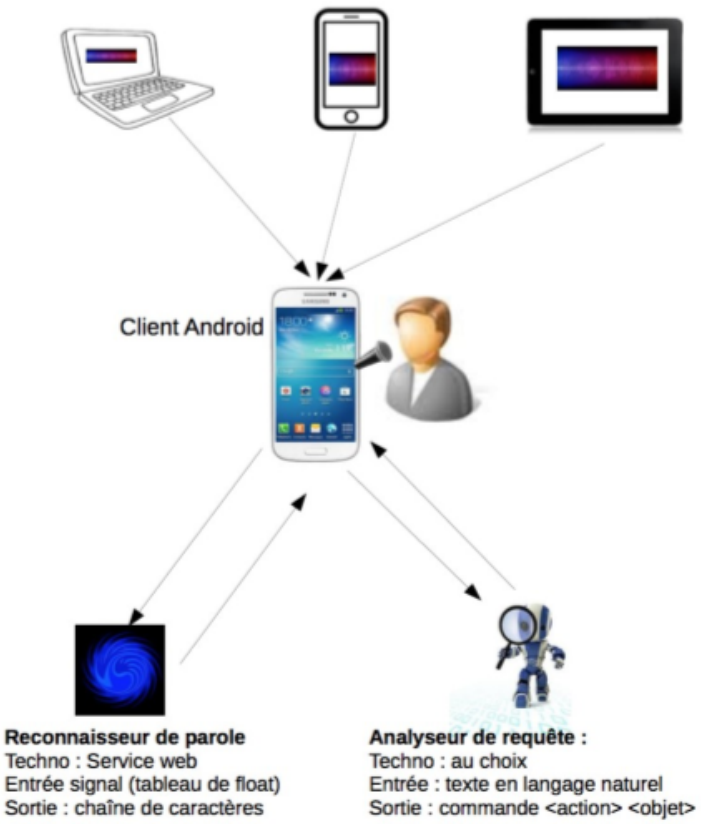

FIGURE 1 – Vision que le client a du système.

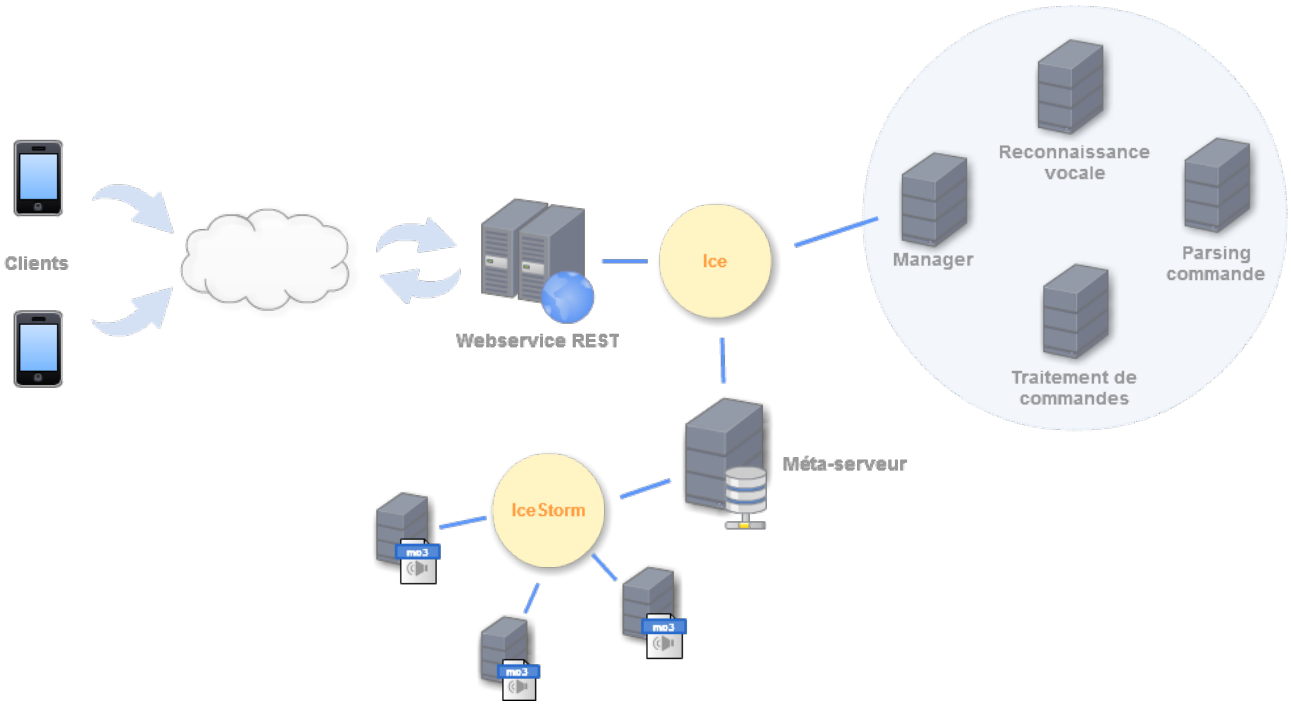

FIGURE 2 – Architecture du système. Sauf le module de Reconnaissance, les autres modules ont été implémentés

2 EXÉCUTION DES MODULES

Serveurs : Pour démarrer le serveur il suffit d'ouvrir un terminal et faire :

- Aller au répertoire <rendu-tp>/Sources/Serveur/src
- Exécuter "make"
- Cela va effectuer la compilation des interfaces Ice et puis lancer les modules IceBox (avec la configuration pour le service IceStorm), Metaserveur, Miniserveur, Manager et WS (webservice).

Tous les messages seront affichés dans le même terminal est stockés dans des fichiers logs séparés par module dans le répertoire "<rendu-tp>/Sources/Serveur/src/launcher/logs".

En ce qui concerne le module Miniserveur, le launcher va lancer deux miniserveurs appelés MINI-1 et MINI-2. Chaque miniserveur sera lancé en indiquant le répertoire où se trouvent les fichiers mp3, "<rendu-tp>/Sources/Serveur/src/miniserveur/musique_1" et ".../musique_2".

Pour arrêter les processus il faut faire Ctrl+C.

Application Android : L'application peut être compilée en ouvrant un terminal et en faisant :

- Aller au répertoire <rendu-tp>/Sources/Client/PlayerDistribue
- Exécuter "./gradlew.bat assembleDebug" (ou ./gradlew pour Linux)
- Cela va compiler télécharger les bibliothèques manquantes, compiler les sources et créer le fichier .apk dans le répertoire "<rendu-tp>/Sources/Client/PlayerDistribue/app/build/outputs/apk".
- L'application peut ensuite être installée dans un smartphone Android.

Sinon, on peut installer directement le fichier APK déjà compilé qui est rendu dans le répertoire "<rendu-tp>/APK".

3 APPLICATION ANDROID

L'application cliente a été développée en utilisant seulement des bibliothèques Android natives. Dans mes tests j'ai utilisé la version 21 du système d'exploitation, mais elle devrait fonctionner dans des version plus anciennes.

Le développement de cette application a représenté un problème du fait que je ne possède pas de smartphone. J'ai utilisé l'émulateur installé avec AndroidStudio. Le plus grand souci avec l'émulateur est qu'il n'arrive pas à accéder aux services Google pour la reconnaissance de la parole. Ce n'est pas clair si l'émulateur n'arrive pas à utiliser le microphone de l'ordinateur ou si c'est un problème dans les applications installés. Après plusieurs recherches sur Internet je n'ai pas été capable d'activer cette fonctionnalité.

Dans la section "Reconnaissance" de ce document j'explique plus en détail les raisons qui m'ont mené à ne pas développer cette fonctionnalité qui était demandée.

Pour simuler la reconnaissance vocale j'ai suivi la méthode recommandée en cours, j'affiche à l'utilisateur une liste de phrases qui représentent plusieurs types de commande que le module de reconnaissance aurait pu retourner. L'utilisateur peut cliquer dessus et le texte est envoyé directement au module de parsing existant dans le serveur centrale.

À travers l'application, l'utilisateur peut :

- "Lister toutes les chansons"
- "Jouer une chanson"
- "Arrêter la chanson actuelle"
- "Envoyer une phrase représentant une commande vocale"

3.1 Détails d'implémentation

Toute la communication avec le serveur est effectuée par des tâches en arrière plan. Un icône est affiché pour indiquer à l'utilisateur que l'on attend la réponse du serveur. Dans l'émulateur, lors d'une commande pour jouer une chanson, on doit attendre plusieurs secondes avant que le MediaPlayer natif commence à lire le streaming.

Le fait d'effectuer des tests chez moi et au CERI, en me connectant et déconnectant des réseaux plusieurs fois, représentait une difficulté. L'adresse IP de mon serveur changeait plusieurs fois et j'étais obligé de reconfigurer l'application Android à chaque démarrage. Pour résoudre cela, la solution que j'ai trouvé consiste à pouvoir me connecter à une adresse fixe en passant par un webservice hébergé dans un serveur public, avec un nom de domaine fixe. J'ai donc créé un site web très simple en Python que j'ai publié sur des serveurs Heroku. Heroku est un site qui permet la publication gratuite d'applications dans le cloud.

L'application Android, lors du démarrage d'une nouvelle requête au serveur, va se connecter au site https ://uapv-m1-s2-util.herokuapp.com/get_util_ip pour récupérer l'adresse IP de mon serveur. De son côté, le serveur de fichiers de musique se connecte à Heroku pour publier son adresse IP lors de son démarrage. Cette solution est imparfaite et non sécure, mais c'était un bon moyen d'éviter trop de configurations de mes applications. En plus, cette solution vous permettra de réaliser des tests de manière très simple.

3.2 Captures d'écran

Ci-dessous je présente quatre captures de l'application en exécution pour montrer les fonctionnalités implémentées.

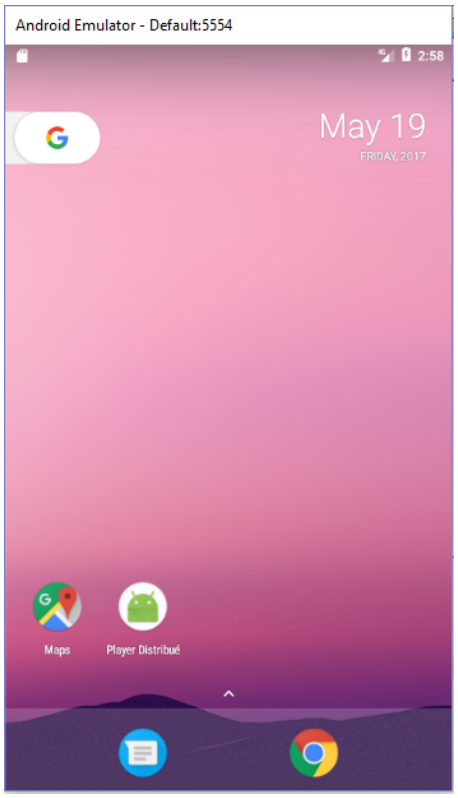

FIGURE 3 – Une fois l'application installée, le logo avec le nom devraient apparaître sur le "bureau".

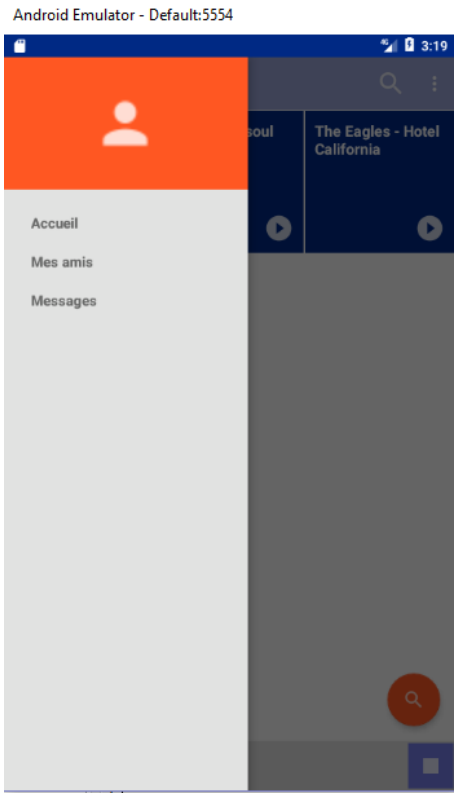

FIGURE 4 – Lors du démarrage, la vue Accueil va communiquer avec le serveur pour récupérer la liste de chansons. Pour recharge la liste, on doit afficher le menu latéral et cliquer sur "Accueil"

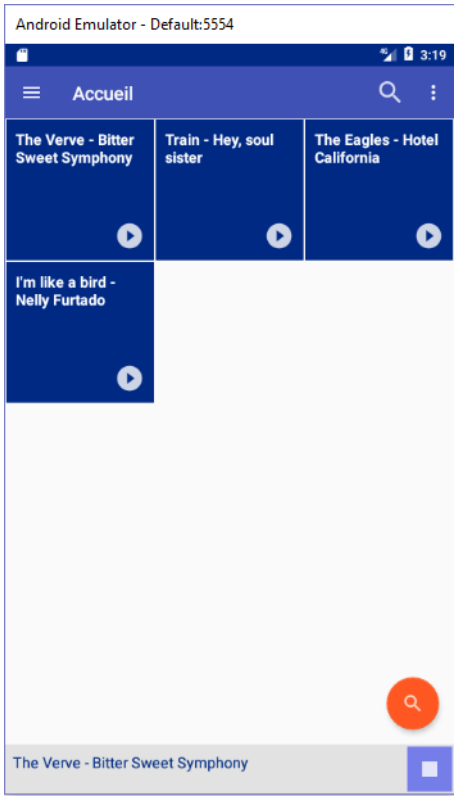

FIGURE 5 – L'accueil affiche un carré pour chaque chanson, on peut cliquer dessus pour jouer la chanson. En bas on voit le nom de la chanson actuelle ou d'autres messages de l'application, à droite le bouton de stop. À droite en bas on voit un bouton orange (FAB) qui permet d'accéder à la liste de phrases.

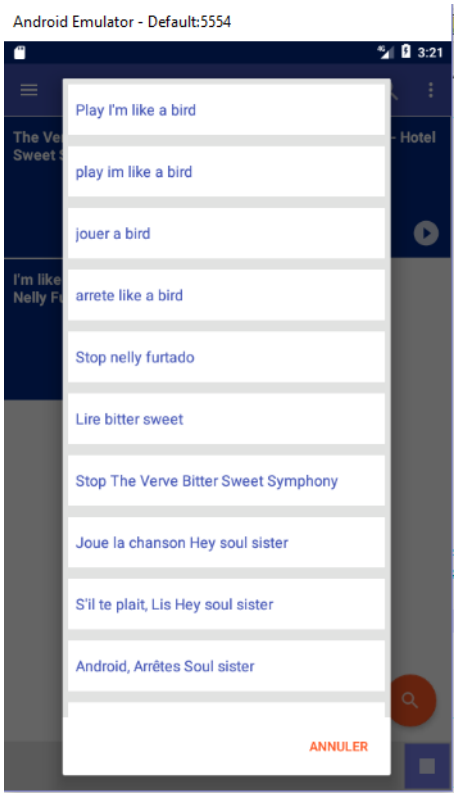

FIGURE 6 – Quand on clique sur le bouton orange, on voit une liste avec plusieurs phrases. En cliquant sur une phrase le popup est fermé et un tâche en arrière plan va communiquer avec le serveur. Pour fermer le popup il faut cliquer sur "Annuler"

4 RÉCEPTION DE COMMANDES UTILISATEUR

Ce module est le point d'entrée à l'application du côté client. Il s'agit d'un webservice REST qui fait appel au module Manager à travers son interface Ice.

Le webservice a été développé en Python en utilisant la bibliothèque web.py.

L'interface du webservice propose l'accès aux URLs suivantes :

- /vocale/(.*)
  Pour cette méthode j'ai uniquement écrit un squelette. Je n'ai pas eu l'occasion de tester la réception de commandes de voix. L'idée était d'utiliser peut-être une liste de floats convertit à string. Un module de Reconnaissance aurait été chargé de lire la data et identifier la phrase prononcée par l'utilisateur.
- /phrase/(.*)
  Cette méthode s'attend à recevoir un string représentant la phrase prononcée par l'utilisateur. Elle est utilisée lors du clic sur les phrases de l'application Android. La phrase est envoyée au module de Parsing, en passant par le Manager, pour qu'il puisse identifier la commande demandée.
- /manuelle/(.*)/(.*) et /manuelle/(.*)
  Les commandes manuelles permettent d'utiliser le système sans être obligé de parler, par exemple dans des endroits trop bruyants. La commande reçoit un paramètre (pour les cas des commande Lister toutes les chansons et Arrêter la chanson actuelle) ou deux paramètres (pour Jouer une chanson).

4.1 Détails d'implémentation

Le webservice démarre par défaut sur le port 30000 du serveur. Cette configuration se trouve dans "<rendu-tp>/Sources/Serveur/src/config/__init__.py".

Toutes les URLs retournent un objet JSON ui permet au client de réagir par rapport au résultat de la commande. Par exemple, s'il reçoit false pour une demande de chanson, il saura qu'il n'y aura pas de streaming musical à consommer.

Le format de la structure de données Commande a changé par rapport au prévu. Actuellement la structure est la suivante :

```
struct Commande {
string commande; //Chaîne représentant la commande à exécuter.
CommandeParams params; //Liste de paramètres pour la commande.
bool erreur; //True s'il y a eu une erreur dans le traitement de la commande.
string msgErreur; //Message de l'erreur ou vide
```

```
string retour; //Représentation JSON pour le retour.
};
```

5 SERVEUR MANAGER

Le serveur manager est à la charge d'orchestrer le traitement d'une commande : appel à un premier module pour reconnaître la phrase, puis au parsing et finalement au module de traitement de la commande. Dans cette version du système, le module de reconnaissance n'a pas été implémenté.

J'ai utilisé le framework Ice, créé par ZeroC.

Le langage de programmation utilisé, comme pour l'ensemble du code du prototype, est Pyhton. Le manager et les autres modules sont implémentés comme des modules Python séparés. Le choix de ce langage et dû principalement au fait que le semestre précédent j'ai trouvé assez simple l'intégration avec Ice et les bibliothèques utilisées.

En ce qui concerne ce module, il n'y a pas un grand changement par rapport à ce qui avait été prévu dans le cahier de charges. Le code fait uniquement l'implémentation de l'interface Ice et puis touts le travail est effectué par les autres modules.

Il faut dire que l'implémentation de la communication en Ice a pris en temps considérable. Les interfaces Ice basiques ne représentent pas un grand problème, sauf le manque de documentation technique. Par contre, j'ai dû passer beaucoup de temps à chercher des informations sur IceStorm pour la communication entre Méta et miniserveurs. Nous étions obligés à utiliser les technologies Ice pour rendre un prototype de cette application pour le cours de Middlewares, j'ai pensé donc que réutiliser le code pour Applications Distribuées était la meilleure manière d'économiser du temps. Malheureusement, le temps passé à bien configurer la communication des modules m'a empêché de m'investir plus sur le module de Parsing.

6 PARSING DE COMMANDES

Le module permet de traiter la phrase prononcée par le client et d'identifier la commande et la chanson concernées. L'idée initiale était d'utiliser la bibliothèque NLTK, mais finalement j'ai utilisé une méthode simplifiée à cause des raisons déjà exposées.

Le module utilise une liste de synonymes pour chaque commande. Par exemple, pour la commande "jouer" on peut identifier des mots comme par exemple "play", "joue", "joues", "lire", etc.

Traitement d'une phrase :

- On commence d'abord par un nettoyage de la phrase reçue. Au début du traitement on passe tous les mots à des minuscules.
- Ensuite, on crée une nouvelle phrase qui ne contient que des lettres ASCII, numéros et espaces. Tous les autres caractères sont ignorés.
- Puis, on crée un tableau avec tous les mots, en séparant la phrase grâce aux espaces.
- On va parcourir le tableau et chercher chaque mot dans la liste de synonymes.
- Si on ne trouve aucun synonyme reconnu, on indique au Manager que la commande n'a pas été reconnue.
- Sinon, on considère deux cas. Soit le mot représentant la commande se trouve à la fin de la phrase (par exemple "Hotel California jouer"), soit le mot se trouve plus ou moins au début.
- Dans le deuxième cas, on dit que peut être l'utilisateur à prononcé une formule du type "S'il vous plaît, jouez Hotel California". Dans ce cas on va ignorer le début de la phrase et prendre tout ce qui se trouve après la commande comme étant le nom de la chanson. Le problème avec ceci est que si on envoie un phrase du style "Jouer la chanson Hotel California", le système va chercher un chanson avec le nom "la chanson Hotel California", et il ne va pas la trouver.
- Si la phrase est reconnue, on retourne une structure de données contenant la commande à exécuter.

La structure de données Commande est celle présente plus haut dans le document.

7 TRAITEMENT DE COMMANDES

Ce module est chargé de recevoir les commandes utilisateur déjà interprétées et d'effectuer les traitements nécessaires. Si c'est une commande pour jouer ou arrêter une chanson, ou pour les lister toutes, il fera suivre la demande au Métaserveur (qui va communiquer avec les miniserveurs, etc). Sinon, il enverra la réponse correspondante au client.

Dans la version actuelle du système, le module renvoie au Métaserveur les commandes pour lister les chansons et pour jouer/arrêter un fichier. Tout autre type de commande n'est pas pris en compte (dans le cahier de charges j'avais prévu pleins de commandes, par exemple pour monter ou baisser le volume).

8 MÉTASERVEUR

Ce serveur est chargé de maintenir une base de données actualisée des chansons existantes dans les serveurs musicaux.
Pour communiquer avec eux j'utilise le framework IceStorm.

Dans la version actuelle, il stocke en mémoire la liste de chansons dans un structure dictionnaire. Les clés du dictionnaire sont les noms des chansons et la valeur est une liste d'objets Chanson. Chaque objet contient toutes les métadonnées du fichier (auteur, nom, genre, etc) ainsi que le nom interne du miniserveur la contenant et le path dans le système de fichiers de ce dernier. Je voulais basculer vers une BDD SQLite, mais je n'ai pas eu le temps de le faire.

Le service propose une interface Ice avec la méthode suivante :

- Appelée par le module de traitement de commandes :
  - traiterCommande(string ipClient, Commande commande)
    Le client peut envoyer seulement deux types de commandes : jouer une chanson ou arrêter la chanson en cours.
    On reçoit en paramètre l'IP du client pour savoir à qui envoyer le flux de streaming musical.
    Lors de la réception d'une commande pour jouer une chanson, on va renvoyer la demande au premier serveur dans la liste de serveurs disponibles.

Topics implémentés actuellement :
Pour la communication Méta-miniserveurs j'utilise actuellement deux topics :

- TopicCommandes
  Le Métaserveur est un publisher dans ce topic, il l'utilise pour envoyer les commandes aux miniserveurs pour récupérer la liste de chansons et pour jouer/arrêter un fichier.
  Les miniserveurs sont des subscribers de ce topic, ils restent à l'écoute de commandes entrantes.
- TopicChansons
  Le but de ce topic est de permettre aux miniserveurs d'envoyer les modifications dans la liste de chansons au Métaserveur. Ils sont des publishers qui peuvent envoyer deux types de messages : liste avec toutes les chansons, string avec l'URL d'un streaming associé à un client particulier (les clients sont identifiés par leur adresse IP).
  Le Métaserveur est un subscriber. Après son démarrage le module reste dans une boucle infinie. Toutes les 10 secondes il va envoyer une commande aux miniserveurs pour récupérer la liste des chansons. Cette procédure est faite en deux parties, d'abord on publie la commande sur le topic TopicCommandes et puis on reçoit la liste grâce au souscripteur de TopicChansons. Le souscripteur va supprimer toutes les chansons associées au miniserveur qui a répondu et insérer les nouvelles.

Limitations du module :

Le système doit pouvoir gérer un ensemble de miniserveurs. Dans ce contexte, le métaserveur doit pouvoir envoyer des commandes à un miniserveur en particulier, et non pas à tous les souscripteurs du TopicCommandes. Par contre, je n'ai pas réussi à envoyer un message à un souscripteur spécifique, je n'ai pas trouvé des techniques comme dans le cas de JMS, où l'on peut définir des filtres sur les messages pour ne recevoir que certaines choses. En fait, pour ainsi faire, d'après la documentation technique de IceStorm, il faut implémenter la logique suivante : le Métaserveur démarre, un miniserveur démarre et demande au Méta d'être inscrit à la liste de souscripteurs, le Méta inscrit le miniserveur et stocke localement une référence à l'objet subscriber créé, ensuite il enverra des messages via cette référence. Cette méthode comportait la modification de beaucoup de choses dans mon architecture, je n'ai pas eu le temps de la mettre en place pour voir si c'est ce dont j'avais besoin.

La nécessité de communiquer avec un miniserveur spécifique apparaît quand on veut jouer une chanson. La méthode actuelle envoie l'objet Chanson via le TopicCommandes. Vu que cet objet contient le nom du miniserveur concerné, chaque miniserveur qui reçoit la commande va comparer son nom propre à celui de la Chanson. Si la commande ne correspond pas à lui, il ne fait rien.

Gestion de la charge des serveurs :

Une chanson pourra exister dans plusieurs serveurs, elle sera identifiée par son nom et sera associée à la liste de serveurs qui la contiennent. Si un client demande une chanson existante dans plusieurs serveurs, on envoie cette demande au premier serveur dans la liste, puis on déplace ce serveur à la dernière place afin d'équilibrer la charge. Ainsi, la prochaine demande pour la même chanson sera envoyée au serveur suivant.

Pour ainsi faire, chaque nom de chanson est associé à la liste d'objets Chanson. Chaque objet représente un fichier dans un miniserveur spécifique. On prend toujours le premier élément de la liste et on le déplace à la fin.

Recherche de chansons :

La recherche d'une chanson est la deuxième partie du parsing de commandes vocales. En fait, une fois que l'on connaît la commande à exécuter, on doit trouver la chanson concernée à partir du nom donné par le client. La technique utilisée est la suivante :

- On fait un nettoyage de la phrase similaire à celui du module de Parsing (on supprime les caractères non Ascii)
- Ensuite, on supprime tous les espaces dans le string représentant le nom de la chanson.
- On compare ce nom avec celui de toutes les chansons dans le dictionnaire (bien sûr, on supprime les espaces dans les noms des chansons avant de faire la comparaison).
- Si on trouve une chanson avec le nom exacte, on retourne celle-là.
- Sinon, on retourne celle dont le nom contient le string cherché ou le string cherché contient le nom. (C'est-à-dire qu'on évalue l'inclusion dans les deux sens)
- Si plusieurs chansons contiennent le string cherché, on retourne celle dont la différence entre les strings est minimale (pour cela on regarde les longueurs des noms).

9 SERVEURS MUSICAUX (MINISERVEURS)

Le système permet à plusieurs serveurs de fichiers de fournir leurs chansons.

Afin de simplifier la gestion des fichiers, je voulais implémenter un processus en arrière plan qui regardait un répertoire spécifique. Les fichiers MP3 existant dans ce répertoire étaient ceux mises à disposition pour les clients. À chaque fois qu'un fichier était ajouté ou supprimé, le thread en arrière plan notifiait le Métaserveur.

(Vous pouvez voir le code pour cette fonctionnalité dans "Serveur/src/miniserveur/demon.py".)

Le problème avec cet approche est que si le Métaserveur tombe en panne, après le re-démarrage il n'aurait plus la bonne version de la liste de chansons. Il faudrait implémenter deux méthodes, la première qui enverrait la liste complète (méthode en place actuellement) et ensuite les mises à jour seraient envoyés directement par le démon. Afin de simplifier l'application, je n'ai gardé que la récupération de la liste entière.

Dans l'idée originale, il y avait en plus une application Python en console qui proposait à l'administrateur quelques options pour s'inscrire/désinscrire du Métaserveur, configurer le répertoire de MP3s et pour avoir accès au log sur les notifications envoyées afin de détecter des possibles erreurs. Cette fonctionnalité n'a pas été implémenté à cause d'une manque de temps.

10 CONCLUSION

Le développement de cette application a été très motivant. Malgré les fonctionnalités non implémentées (notamment la reconnaissance vocale), j'ai investi beaucoup d'heures dans le développement. Je trouvait très intéressant la possibilité de mettre en pratique toutes les choses que l'on a appris cette année et de prendre le temps de faire une première application Android. Le bilan est donc positif.

Néanmoins, je trouve dommage que l'application ne corresponde qu'à une seule matière. Il aurait été intéressant d'avoir la possibilité de travailler avec plusieurs professeurs sur le même sujet, ce qui nous aurait permis d'avoir plus de temps pour bosser dans ce projet au lieu de devoir diviser nos heures de travail dans plusieurs application similaires. Par exemple, pour cette matière nous avions la liberté d'utiliser n'importe quelle technologie de messagerie. Par contre, pour le cours de Middlewares nous étions obligés d'utiliser Ice, avec les problèmes déjà présentés d'utiliser cette technologie .

Enfin, j'aurais bien apprécié avoir eu le temps de m'investir plus en profondeur dans la reconnaissance vocale et le traitement des phrases. Par exemple, une solution aux commandes non reconnues par le module de Parsing aurait été d'utiliser une sorte de base de connaissances. À chaque commande non reconnue le système aurait pu demander à l'utilisateur d'indiquer la commande souhaitait, cette réponse pourrait être stockée dans une BDD et utilisée par la suite.