Rapport application

Middlewares

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 faut développer une application distribuée qui permettra à des clients Android de jouer des fichiers de musique. Les chansons se trouveront dans un ensemble de serveurs distants.

Le système comportera plusieurs modules :

- Application cliente Android
- Un webservice qui recevra les commandes utilisateur.
- Un serveur central qui va orchestrer l'accès à tous les modules
- Un module de reconnaissance de la parole
- Un module de parsing de commandes
- Un module de traitement de commandes
- Un méta-serveur qui connaîtra la liste globale de chansons
- Un ou plusieurs serveurs de fichiers musicaux (miniserveurs)

Pour le cours de Middlewares, la version rendue de l'application est un prototype dont le but principal est de montrer la communication interne entre les composants du serveur, notamment les échanges entre le Métaserveur et les miniserveurs.

La technologie utilisée pour la communication est essentiellement Ice by ZeroC. Chaque module serveur une interface Ice et les échanges entre le Métaserveur et les miniserveurs est faite vie IceStorm et IceBox.

Modules non implémentés dans cette version :

Dans ce prototype, l'application Android a été remplacée par un client en console très simplifié, qui permet le listing des chansons, le démarrage et l'arrêt de fichiers musicaux.

Le webservice qui se comporte comme une façade pour le système n'est pas utilisé. Le client accède directement au Manager via son interface Ice.

En outre, le module de reconnaissance vocale n'a pas été utilisé du fait que le client en console ne peut pas y accéder.

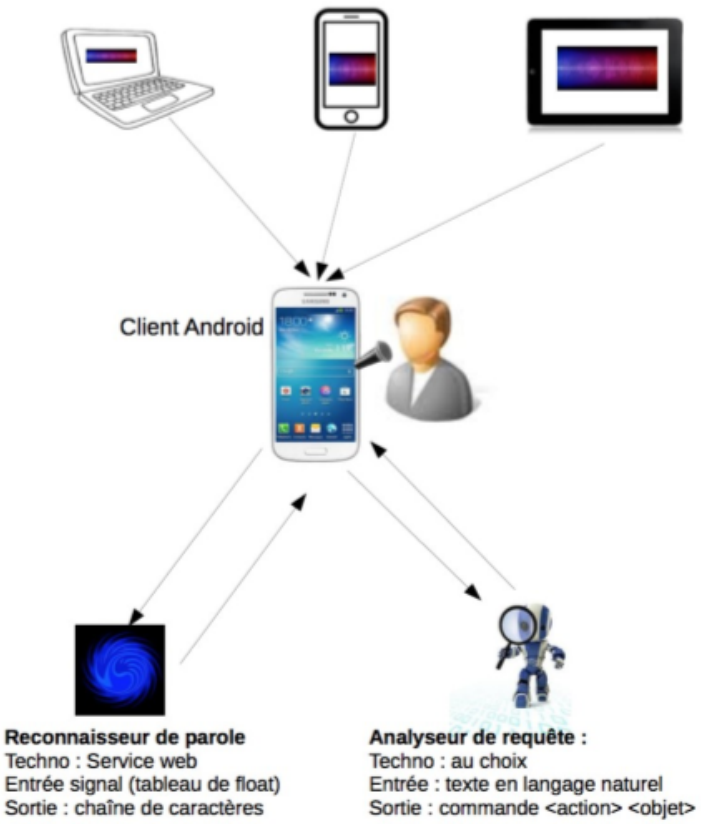

FIGURE 1 – Vision que le client a du système.

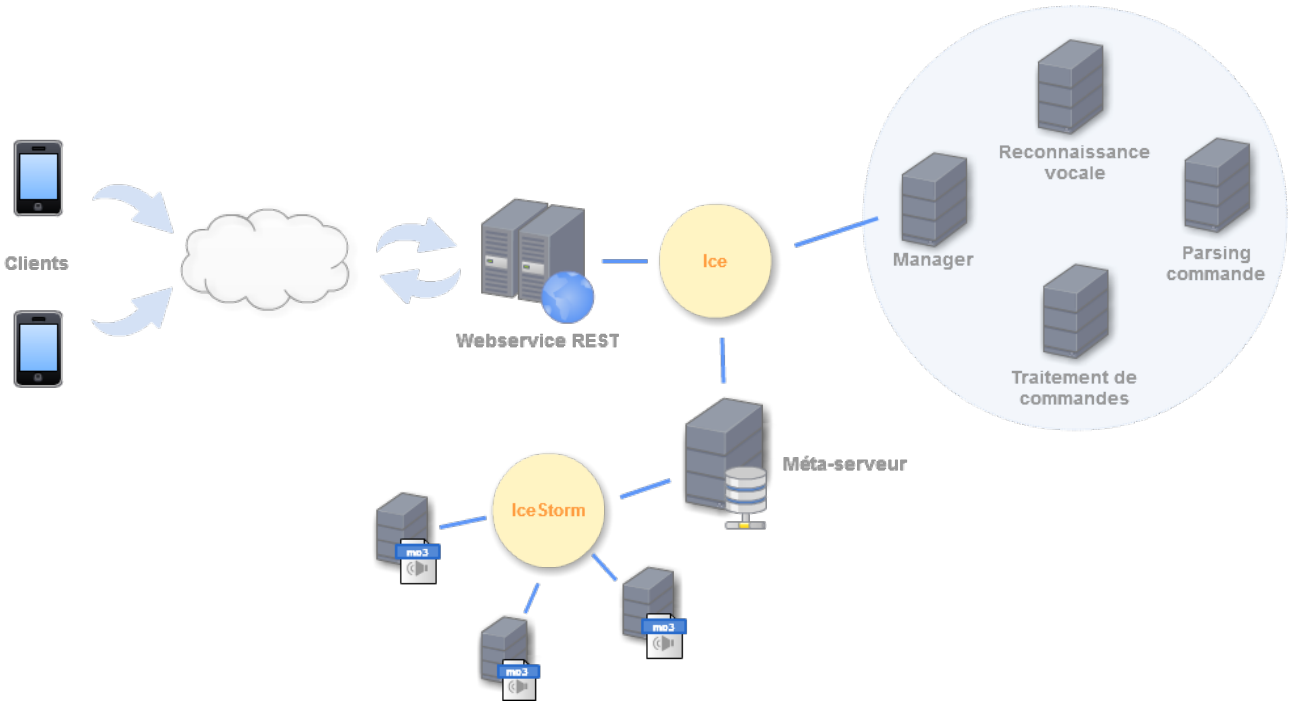

FIGURE 2 – Architecture du système.

2 EXÉCUTION DU PROTOTYPE

Pour exécuter le prototype il faut ouvrir cinq terminales : Client, IceBox, Manager, Métaserveur et ou moins un Miniserveur. Pour la version finale, l'idée est de créer un launcher qui afficherait dans une même terminale tous les messages des autres modules. Même si le système doit fonctionner dans des machines différentes, pour faire du test il est plus pratique de lancer tout dans une même console.

Pour lancer les différents modules (il faut lancer d'abord IceBox, puis Métaserveur et puis le reste) :

- IceBox
  - Aller au répertoire <rendu-tp>/Serveur/src
  - Exécuter "make run-icebox"
  - Cela va lancer IceBox avec la configuration pour le service IceStorm.

- Métaserveur
  - Aller au répertoire <rendu-tp>/Serveur/src
  - Exécuter "make build" (une seule fois et ça marche pour tous les modules) pour compiler les interfaces Ice
  - Exécuter "make run-meta"
  - Cela va lancer le module du Métaserveur et va se connecter à IceStorm. Ensuite, le module entre dans une boucle pour demander la liste des chansons aux miniserveurs.

- Manager
  - Aller au répertoire <rendu-tp>/Serveur/src
  - Exécuter "make run-manager"
  - Cela va lancer le module du Manager, qui va gérer les étapes de traitement d'une commande.

- Miniserveur
  - Aller au répertoire <rendu-tp>/Serveur/src
  - Exécuter "make run-mini"
  - Cela va lancer le module d'un miniserveur et va se connecter à IceStorm.
  - ATTENTION, pour ce module on a besoin de configurer le répertoire qui contiendra les fichiers musicaux. La configuration se trouve dans le moule config, dans le fichier <rendu-tp>/Serveur/src/config/__init__.py. Modifier la variable MINISERVEUR_SONGS_PATH. Par défaut, le répertoire utilisé est <rendu-tp>/Serveur/src/miniserveur/musique.

- Client
  - Aller au répertoire <rendu-tp>/Client/ClientConsole/src
  - Exécuter "make"
  - Cela va compiler les interfaces Ice du côté client et lancer l'application en console. Un menu très simple vous guidera dans l'application.

3 APPLICATION CLIENTE

L'application cliente a été développée en Python.

Ce client permet à l'utilisateur de connaître les fichiers musicaux existants dans l'ensemble de serveurs. On peut choisir une chanson et la jouer/arrêter avec des commandes manuelles. La chanson sera reçue en forme de streaming. Pour ainsi faire, j'utilise la bibliothèque VLC.

Les commandes disponibles sont :

- "Jouer une chanson"
- "Arrêter la chanson actuelle"
- "Lister toutes les chansons"

4 SERVEUR MANAGER

Dans la version finale du système, un webservice REST sera chargé de recevoir les demandes provenant de l'application Android et de les transmettre au Manager. Dans ce prototype, le Manager reçoit les commandes directement à travers son interface Ice.

Le serveur manager sera 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. Pour le prototype, les modules de reconnaissance et de parsing ne font rien en particulier.

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.

5 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, il fera suivre la demande au Métaserveur. Sinon, il enverra la réponse correspondante au client.

Dans l'état actuel, le module renvoie au Métaserveur les commandes pour lister les chansons et pour jouer/arrêter un fichier.

6 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. Ce module va s'inscrire à un service de messagerie IceStorm pour recevoir les mises-à-jour des fichiers musicaux dans les serveurs distants. De leur côté, ces derniers seront inscrits à IceStorm en tant qu'émetteurs.

Dans la version finale, il va stocker la liste de serveurs existants et la liste globale de chansons, en faisant le lien entre les chansons et le miniserveur qui les contient. Pour ainsi faire, je vais d'abord utiliser des structures en mémoire et ensuite basculer vers une BDD SQLite.

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 trois types de messages : liste avec toutes les chansons, ajouter une seule chanson, supprimer une seule chanson de la liste.
  Le Métaserveur est un subscriber. Il implémente une boucle dans laquelle il va envoyer toutes les 10 secondes une commande listerChansons (via le TopicCommandes) pour demander aux miniserveurs de répondre avec leurs fichiers. Ensuite, il va recevoir les réponses grâce à un subscriber inscrit au TopicChansons.

Limitations du prototype :

Le système finale devra pouvoir gérer un ensemble de miniserveurs. Dans ce contexte, le métaserveur devra pouvoir envoyer des commandes à un miniserveur en particulier, et non pas à tous les souscripteurs du TopicCommandes. 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 au subscriber créé, ensuite il enverra des messages via cette référence.

Le prototype actuel a la limitation de ne pas gérer un ensemble de miniserveurs, le Méta envoie de messages à tout le topic et tous les miniserveurs actifs recevront la demande. Dans le code vous trouverez un troisième topic TopicMiniserveurs qui permettrait de résoudre ce problème. Les miniserveurs publieraient dans ce topic pour indiquer leur arrivées et départs. ***Pour le but de ce TP, vu que l'objectif est de prendre en main les outils et de comprendre la logique des middlewares orienté messages, je n'ai trouvé pas nécessaire de complexifier le code actuel pour gérer ce cas. Je pense que le prototype montre bien les interactions Publisher/Subscriber. Néanmoins, je le ferai pour la version finale du système.***

Une deuxième limitation est le cas de bord où le Métaserveur démarre après les miniserveurs. Pour gérer ce cas, j'estime que la solution idéale serait d'utiliser un quatrième topic (ou réutiliser un autre en ajoutant les messages nécessaires) où le Méta publierait un message de broadcast à tous les miniserveurs pour leur indiquer son démarrage et leur demander de s'inscrire au TopicCommandes. Ce cas n'est pas géré par le prototype.

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 enverra cette demande au premier serveur dans la liste, puis on déplacera ce serveur à la dernière place afin d'équilibrer la charge. Ainsi, la prochaine demande pour la même chanson sera envoyée au serveur suivant.

Cette fonctionnalité n'est pas implémentée dans le prototype, elle est liée à la première limitation citée ci-dessus.

7 SERVEURS MUSICAUX (MINISERVEURS)

Le système permettra 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 redémarrage il n'aurait plus la bonne version de la liste de chansons. Il faudrait implémenter deux méthodes, la première qui chargerait toute la liste depuis les miniserveurs (méthode en place actuellement) et ensuite les mises à jour seraient envoyés directement par les miniserveurs. Afin de simplifier le prototype, 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 pourrait être reprise pour la version finale si le temps est suffisant.