CHITIMBO Manyanda Charles
LARBI YOUCEF Mohamed Reda

Programmation concurrente, réactive répartie
Jammin Session

I- Introduction

L'objectif de ce projet est d'implémenter un client et un serveur, dans deux langages différents, permettants à plusieurs musiciens de jouer à distance. Le serveur s'occupe de mélanger les sons provenant de chaque client et de renvoyer à ceux ci le son mélangé.

II- Choix des langages

Une des particularités de cette application concerne les langages d'implémentation. En effet, celle-ci utilise deux langages de programmations différents, un pour le client, un pour le serveur.

Nous avons choisi Java implémenter le serveur, et C pour le client. Nous avons choisi ces langages car ce sont des langages que l'on connaît bien, en plus d'être des langages plutôt différents (Sinon nous aurions pris Java/C++). Une fois les langages choisis, nous avons donné au serveur le Java afin de simplifier son écriture, le Java ayant des outils plus haut niveau que le C.

III- Environnement et outils exterieurs

Le système d'exploitation sur lequel sera implémentée cette application sera Linux. Nous avons donc testé ce programme uniquement sous Linux.

Concernant les outils externes permettant d'exécuter le programme, nous avons besoin:

Pour le serveur : De l'outil *ant* afin de compiler, et exécuter le programme.

Pour le client : De la librairie *ALSA*, permettant d'enregistrer et de jouer du son sous Linux. Nous avons en particulier besoin du paquet de développement, nommé *libasound2-dev.*

IV- Extensions

Concernant les extensions, nous avons implémenté le chat. Les clients peuvent communiquer en ecrivant et en recevant des messages dans la console.

V- Architecture du serveur

Le serveur est implémenté en Java 8. Cette implémentation décompose l'architecture du serveur en 7 classes Java placées dans 3 packages ; Mains, Servers, Tools.

Le package Mains contient une classe qui prends les arguments ; dans notre cas les options de la ligne de commande lors du lancement de notre application. Cette classe permet de traiter ces options pour savoir le nombre maximum de clients que peut contenir notre serveur, le port où le serveur sera lancé, ainsi que le timeout. Si aucune de ces commandes ne sont passées le serveur se lancera avec les options par défauts. C'est le cas aussi lors de la présence de fautes dans une option, alors la valeur de celle si sera celle par défaut. Cette même classe contient la méthode main pour lancer notre serveur.

Le package Tools contient les classes utilitaires pour notre implémentation. On a notamment la classe AudioChunk qui modélise un buffer audio avec son tick correpondant. On a une classe qui nous produit un nom d'utilisateur unique. Et la derniere classe dans ce package, HardCodedValues, contient toutes le protocole utilisé dans notre application. En plus du protocole, cette classe contient toutes les constantes de notre application.

Le package Servers contient deux classes et son tour. La première classe Server modélise un serveur en contenant toutes les informations actuelles de notre système I.e les clients connectés et leur état courant ; le tick courant, style et tempo, s'il est vide, non vide ou complet, accepter la connexion de la part des clients etc . Cette classe a aussi une méthode pour le mixage du flux audio des musiciens. Pour mixer deux buffer audio on a utilisee la formule

pour tous I a partir du 0 a taille du buffer

result[i] = buffer1[i] + buffer[2] >> 1

Le serveur peut envoyer un message sur les flux de connexion de tout le monde ou

de tout le monde a l'exception de celui qui demande. On peut aussi demander à notre serveur un buffer audio à un client pour un tick donné. On a prévu aussi une demande de fermeture des canaux suite à une demande des clients.

Un client est modélisé par une classe a part ServiClient qui s'occupe de créer les deux canaux ; connexion et data, d'envoyer ces canaux aux serveur et de demander au serveur s'il le jam peut se mettre à jour ou pas. Dans cette classe, la gestion du timeout est aussi faite. Le client maintient son tick courant et avec ce tick, il peut demander au serveur de lui envoyer le buffer audio décalée d'un certain temps relatif.

Ces deux classes dans ce packages sont des classes sont tous des threads Java.

IV- Architecture du client

Pour implémenter le client, on décompose celui ci en 8 modules :

- *Commande handler* : Ce module permet de gérer les commandes, d'entrée ou de sortie. Il a pour principal rôle de convertir une commande sous forme de chaîne de caractères en une structure représenant une commande que l'on définit. Une commande est notamment identifiée par un identifiant, qui n'est autre que le nom de la commande, ainsi que ses arguments. Afin de simplifier les manipulations, on représentra l'identifiant sous forme d'une enumération. On associe donc un entier à chaque nom de commande.

  Ce module permet donc de créer une commande, en prenant en compte les caractères d'échappement, et en vérifiant, pour chaque commande, qu'elle comporte le bon nombre d'arguments

- *Connection* : Ce module permet de gérer la connexion au serveur, l'envoi et la récéption de messages, et la deconnexion. Ce module a pour principal but de simplifier l'utilisation de ces fonctionnalités, en les rendant plus haut niveau.

- *Sound* : Ce module permet de gérer les flux audio. Il a notamment pour but de créer un gestionnaire de flux audio, d'entrée ou de sortir, de jouer ou de capturer un son, selon le type du gestionnaire audio, et enfin de détruire un gestionnaire de flux audio. L'implémentation de ce module est basée sur la bibliothèque *ALSA*.

- *Session :* Ce module permet principalement de représenter une session, qui est caracterisée par le tempo, le nombre de musiciens, et le style. Ce module permet donc de c

- *Tools :* Ce module permet de gérer des fonctionnalités utilitaires comme le parsing des arguments passés en ligne de commandes pour lancer le programme, ou encore de lire sur stdin de façon sûre.

- *Input :* Ce module permet de gérer l'entrée du client. Il permet, pour chaque commande, d'efféctuer le traitement adéquat.

- *Output :* Ce module permet de gérer la sortie du client. Il permet notamment d'envoyer les différentes commandes possibles du client.

- *Core :* Ce module est le module principal. Il permet de gérer le coeur du client. Pour cela, on représente celui-ci par une structure qui contient divers informations sur le client, comme entre autres, les deux sockets de communication, la session, les deux gestionnaires de son, le nom de l'utilisateur etc.

  Le principale rôle de ce module est de lancer le client. Le déroulement du client se déroule en deux étapes : la phase de connexion au serveur, et la phase principale qui consiste à envoyer continuellement un flux audio enregistré, et attendre les réponses du serveur (quelque soit la réponse), et enfin la phase de destruction du client. De plus, la phase principale contient aussi la gestion du chat, qui affiche sur la console tout nouveau message, et envoi tout message ecrit sur *stdin*. A la fin de la phase principale, le client est détruit et toute les ressources associées sont libérées.

  A noter aussi que le core du client contient 3 threads : Un pour la capture de l'audio, un pour la lecture du flux audio, et un qui gère le protocole.

VII- Tests

Pour tester le client, nous avons fait un programme de test pour chaque module (hormis les modules *input* et *core*). Pour le module de son, nous avons fait deux programmes de test, un pour l'enregistrement, un pour la lecture.

Nous n'avons pas rencontré de problèmes majeurs avec tests, excépté avec le module de son, et particulièrement la lecture du son. En effet, lorsqu'on lance le programme, celui-ci fonctionne et on entend bien ce qu'on a enregistré auparavant, mais lorsqu'on le relance une deuxième fois tout de suite aprés, nous avons des «underrun error» et on entend des sons nuisibles. Il faut donc attendre pendant un moment pour pouvoir le relancer. Nous n'avons pas réussi à résoudre ce problème, qui a priori, est un problème de buffer non lu entièrement, ce qui provoque ce problème.

Enfin concernant les tests sur le programme entier, tout fonctionne relativement bien. Nous avons cependant quelques problèmes. Le premier concerne l'envoi du buffer contenant le son. En effet, celui ci contient des données binaires mais il est envoyé sous forme de chaîne de caractères. Il se peut donc que ce buffer contient des caractère de saut de ligne, comme '\n' ou autre, en plein mileu. Or le serveur attend une ligne entière, ce qui peut provoquer la réception de buffer incomplet. Pour remédier à cela, nous avons fait une fonction qui permet de «nettoyer» le buffer aprés que celui-ci ait été capturé par le microphone. Cette fonction consiste à remplacer tout les caractères «parasites» comme les '\n' par des caractère proches qui ne posent pas de problème. Le buffer est certes modifié, mais ce n'est pas génant étant donné que l'on entend aucune différence, à priori.

Nous avons aussi un problème de synchronisation. Il y'a un client qui est plus rapide que les autres. Au point qu'on obtient un retard assez conséquent pour certains clients.

Autre bug à noter, lorsqu'on lance le client, il est possible que celui-ci reste bloqué sur «Trying to connect...». Il faut donc relancer le client.