Network musical jammin

Projet PC2R - 2015

AFFES Mohamed Amin, KOBROSLI Hassan - 26 avril 2015

Introduction

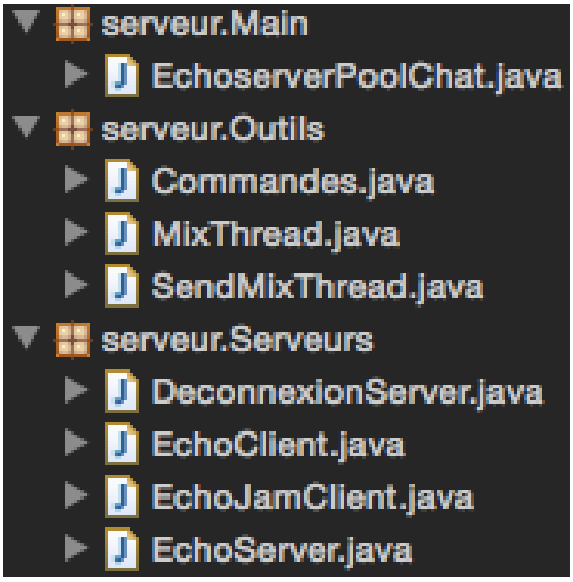
Pour ce projet, nous réalisons une application de jam session permettant la connexion de plusieurs clients à un seul serveur principal commun recevant les sons joués et envoyant le mélange des sons des autres clients à chacun des clients afin que le son produit finalement par chaque client soit un mélange de tous les sons des clients connectés donnant l'impression qu'ils jouent au même endroit.
La plus grosse problématique dans ce projet est la synchronisation des sons reçu coté serveur en temps réel (pas tellement on décale les sons émis de 4 temps), le temps pris en compte ici est le temps logique musical représenté par les temps (ticks) envoyés par le client.
Concernant la partie bonus nous avons réalisé la partie « Discussion instantanée ».

Présentation de la hiérarchie des classes:

1-Serveur

Concernant le choix du langage pour le serveur, le choix s'est porté sur le langage Java, la raison de ce choix est tout d'abord la gestion de la synchronisation entre des Threads concurrents qui est je trouve plus simple d'utilisation par rapports à d'autres langages. Ainsi que l'utilisation d'une bibliothèque effectuant la synchronisation (mélange) entre plusieurs fichiers audio.

Voici la hiérarchie des classes concernant le serveur :



Nous pouvons voir donc l'existence de deux packages (audio et serveur), le deuxième contenant à son tour trois packages (Main, Outils et Serveurs).

- **Le package « Main »** contient une seule classe qui est la classe démarrant le serveur de l'application, Le serveur doit prendre en paramètre le nombre de musiciens maximal, la durée du timeout (en millisecondes) après laquelle un musicien n'est pas considéré dans le flux audio. par défaut le nombre maximal de musiciens vaut 4, la durée du timeout 1000 millisecondes et le port 2015.

- **Le package « Outils »** contient trois classes :
- La classe « Commandes.java » contient des méthodes statiques réalisant l'envoi des différents protocoles donnés dans l'énoncé pour un ou plusieurs clients (cas du chat), elle contient aussi une méthode statique « filter » supprimant les caractères spéciaux d'une chaîne de caractères.
- La classe « MixThread » est la classe représentant le Thread s'occupant des mélanges il prend une liste de buffers pour un client précis, effectue le mélange pour ce client comme décrit dans l'énoncé du projet (addition des buffers et faire attention aux dépassement des bornes (-1,1)) , ajoute le buffer contenant le mélange à la file d'attente du client puis notifie le Thread s'occupant de l'envois du mélange au client.
- La classe « SendMixThread » s'occupe de l'envoi du buffer de mélange au client en le retirant de la file d'attente. Il attend la notification du Thread s'occupant du mélange puis attend que le tick actuel soit égal au tick correspondant au moment de l'envoi au client, puis récupère le buffer de la file d'attente et l'envoie au client, puis il vérifie si tous les buffers correspondant à son tick sont envoyés aux différents clients, si c'est le cas les données correspondant à ce tick sont supprimés.

- **Le package « Serveurs »** contient quatre classes :
- La classe « DeconnexionServer » représente un Thread lancé au début du serveur qui attend l'entrée du mot « exit » sur l'entrée standard pour déconnecter le serveur et fermer les sockets.

- La classe « EchoClient » est un Thread gérant un client. Il attend d'être notifié de la connexion d'un client, vérifie s'il est concerné par cette connexion, c'est à dire si la connexion est sur le socket principal de protocole représentant la connexion d'un nouveau client à la session. Si c'est le cas et si la capacité de la session n'a pas atteint sa borne maximale, le Thread récupère le socket de la connexion avec ce client et prévient le serveur de la connexion. Il attend ensuite de recevoir les protocoles clients et appelle une méthode du serveur traitant ces demandes. Si la demande est inconnue, aucun traitement n'est effectué. Si le client envoie un message vide ou se déconnecte ou envoie un protocole de demande de déconnexion, le serveur le déconnecte proprement puis se remet en attente d'un nouveau client. Le déroulement des protocoles et l'ordre est défini dans la méthode de réponse au client qui est détaillée dans la présentation de la classe « EchoServer ».

- La classe « EchoJamClient » est un Thread gérant un client se connectant au socket de gestion du son, il attend d'être notifié de la connexion d'un client, vérifie s'il est concerné par cette connexion c'est à dire si la connexion est sur le socket de son. Si c'est le cas, le Thread récupère le socket de la connexion avec ce client et prévient le serveur de la connexion. Ensuite, le serveur attend un message du client, si un délai timeout fixé est dépassé avant la réception du message ou si le message ne correspond pas au protocole attendu de réception du buffer, le client reçoit un message du protocole audio_ko puis le serveur le déconnecte des deux serveurs (principal et audio). Sinon, il récupère le buffer et le tick puis ajoute le buffer à la HashMap principale contenant les buffers ayant comme clefs leurs ticks, puis attend la réception des buffers des autres clients correspondant au même tick ensuite il lance les deux Threads s'occupant du mélange et de l'envoie du résultat au client.
- La classe « EchoServeur » est la classe représentant le serveur principal, ce serveur crée un socket serveur sur le port indiqué par l'utilisateur (ou 2015 par défaut). Dans le constructeur on crée et lance un ensemble de threads que l'on conserve dans une liste (*Vector*). Les threads clients se mettent en attente d'un signal du serveur. Lors d'une nouvelle connexion, le serveur ajoute le nouveau socket dans une liste d'attente et notifie un des threads. Le thread doit alors se charger de récupérer la connexion du nouveau client et d'effectuer le comportement adéquat. Nous utilisons donc ce que l'on appelle un système de Pool de Threads. Cette classe contient aussi des méthodes qui sont appelées lors d'une nouvelle connexion d'un client ou de sa déconnexion ou d'autres accesseurs à des variables permettant la bon fonctionnement du serveur et la manipulation des champs contenu dans cette classe, notamment les buffers audio, les sockets, le nombre de clients connectés au serveur, la capacité maximale, etc. La méthode « writeAllButMe » envoie le message d'un client aux autres clients connectés (utile donc pour le chat). Les deux méthodes principales de cette classe sont « AnswerClient » et « AnswerJamClient ». La première est appelée à chaque fois qu'un client envoie un message. Elle vérifie la correspondance du message envoyé avec les protocoles connus et effectue dans le cas d'une correspondance la tache demandée. Si on prend l'exemple de la connexion d'un client par exemple, le serveur va enregistrer ce nom d'utilisateur lui envoyer un message de bienvenue puis lui demander de saisir les options de la session Jam s'il s'agit du premier client, ou l'informe des options sinon. Le client est ensuite informé du port de connexion audio, puis l'informe de la réussite de la connexion dès que celui-ci se connecte, puis démarre un Thread s'occupant de cette connexion. La deuxième méthode est celle appelée par le serveur lors de la réception d'un message provenant du client sur le socket audio. La méthode récupère le tick et le buffer envoyés par le client ou retourne le buffer ou un « null » dans le cas d'une erreur.

L'ensemble de ces classes forme notre serveur s'occupant de gérer les connexions et déconnexion des clients et de l'échange Client/Serveur coté Serveur.
Pour résumer, notre serveur attend donc la connexion des clients qui ne doit pas dépasser une borne max définie, puis dès qu'un client se connecte il attend le message de connexion et lui donne les informations nécessaires pour qu'il puisse se connecter au port audio. Les données audio reçues sont stockées par le serveur dans la HashMap, mélangées dès réception de tous les buffers pour un tick précis. Le serveur ajoute le résultat à une file d'attente puis l'envoie au client dès que le tick actuel est égal au tick auquel il doit être envoyé. Rien n'est envoyé pour les 4 premiers temps. La discussion instantanée est effectuée sur le socket principal selon le protocole défini dans l'énoncé.

2-Client

Le client est programmé en C. Il utilise l'API Alsa pour la partie capture et playback audio. Alsa est le driver par défaut de la carte son sur la plupart des distributions linux. Le programme enregistre et envoie les sons sur les périphériques d'enregistrement et de lecture par défaut (modifiables dans les paramètres système).

La première partie du client concerne la connexion au canal de contrôle. Cette connexion ce fait sur le thread principal (main). C'est sur cette connexion qu'on envoie et reçoit les commandes. Les commandes reçues sont analysées et traitées en conséquence.
Si le client est le premier connecté au serveur, il choisit le style de musique et son tempo.

Dès réception du port de connexion audio, un deuxième thread est démarré. Ce thread s'occupera de toute la partie audio. Il démarre deux autres threads, le premier pour la capture audio, et le deuxième pour la réception du mix des autres joueurs provenant du serveur. Ces deux threads disposent chacun d'un canal de sortie vers la carte son.
Si le client est le premier connecté au serveur, le thread de capture est démarré immédiatement. Sinon, il est envoyé dès réception du premier buffer mix, ceci afin que la capture et la lecture soient synchronisées.

Le client et le serveur s'échangent des buffers contenant des valeurs entières codées sur 16 bits (entre -32767 et 32767). Cela permet principalement d'envoyer des paquets plus petits qu'avec des flottants (chaque entier prend au maximum 6 caractères), même si il est vrai que la précision est moindre.

Les buffers envoyés au serveur correspondent à un temps (la taille est calculée par la formule size = fréquence d'échantillonnage * 60 / tempo). Le buffer du temps 0 est donc envoyé au moment où débute la capture du temps 1. Chaque buffer est envoyé via un nouveau thread, pour éviter tout blocage de l'enregistrement.

Le buffer qui sert à la capture est de taille plus réduite (size / 100), pour éviter d'avoir une latence trop grande entre le moment de la capture et le playback.

Nous avons noté quelques problèmes en utilisant une fréquence d'échantillonnage de 44.1 kHz (sons déformés).

Les buffers de mix envoyés par le serveur ont une taille de 1024.

Si l'utilisateur ne souhaite pas capturer de son depuis l'entrée de la carte son (et aussi à des fins de test), il peut utiliser l'option [-f] du programme qui permet de lire un fichier audio à la place. Cette fonctionnalité utilise l'API Libsndfile. La liste des formats supportés est fournie en annexe.

Annexe

Formats des fichiers audio supportés :

Microsoft WAV format (little endian).
Apple/SGI AIFF format (big endian).
Sun/NeXT AU format (big endian).
RAW PCM data.
Ensoniq PARIS file format.
Amiga IFF / SVX8 / SV16 format.
Sphere NIST format.
VOC files.
Berkeley/IRCAM/CARL
Sonic Foundry's 64 bit RIFF/WAV
Matlab (tm) V4.2 / GNU Octave 2.0
Matlab (tm) V5.0 / GNU Octave 2.1
Portable Voice Format
Fasttracker 2 Extended Instrument
HMM Tool Kit format
Midi Sample Dump Standard
Audio Visual Research
MS WAVE with WAVEFORMATEX
Sound Designer 2
FLAC lossless file format
Core Audio File format
Psion WVE format
Xiph OGG container
Akai MPC 2000 sampler
RF64 WAV file

API :

http://www.mega-nerd.com/libsndfile/api.html
http://www.alsa-project.org