Musical jam session server

Dans cette exercice, nous allons réaliser un serveur permettant d'effectuer des jams sessions en temps-réel entre des musiciens répartis à travers le monde. Le principe d'une jam est que les musiciens se synchronisent sur un tempo commun puis se mettent à improviser en fonction du jeu des autres. Nous allons donc modéliser le système par un serveur TCP/IP gérant les différents musiciens qui chacun possèdent un client permettant d'enregistrer et écouter l'ensemble simultanément. Un tel système est représenté dans la figure 1

Nous considèrerons dans cet exercice qu'il n'existe qu'une seule jam par serveur (tous les clients jouent tous ensemble). Dans la première partie de l'exercice, nous considèrerons également que la communication est *centralisée*, ie. le serveur gère toutes les interactions. Au démarrage, il se comporte comme un serveur classique en attente de client. Lorsqu'un nouveau client se connecte :

- Le serveur crée un nouveau thread de communication avec le client
- En fonction du nombre de musiciens, le serveur répond à la connexion par des messages différents
  - Si la session est vide, le serveur demande au premier musicien de régler les informations de *style* et *tempo*
  - Si des musiciens sont déjà connectés, le serveur envoie les informations de base de la jam session (*style*, *tempo, nombre de musiciens connectés*)
  - Si le nombre maximum de musiciens est atteint, la connexion est fermée proprement
- Il envoie ensuite au client un port sur lequel il va se mettre en attente pour ouvrir un second canal (dédié au flux audio)
- Le client se connecte au second port pour établir la connexion
- Le serveur répond par le canal de contrôle pour confirmer l'ouverture du canal audio

Une fois la connexion établie, la boucle d'interaction fait en sorte de gérer les flux audio et de synchronisation.

- Chaque client enregistre en permanence l'instrument local et envoie régulièrement des paquets audio au serveur
- Le serveur reçoit les différents flux audio et s'occupe de les synchroniser et de les mélanger (à noter que le serveur prépare un mélange spécifique pour chaque client contenant le flux audio mélangé de tous *sauf le sien*).
- Le serveur envoie périodiquement le flux audio mélangé spécifique à chaque client.

On considèrera que l'on possède la magnifique librairie multi-langage Audio permettant d'enregistrer, recevoir et écouter des buffers audio grâce aux fonctions suivantes

`void initAudioRecording(float *audioBuffer, int buffSize, pthread_cond_t *nextBufferIsReady);`

Permet l'initialisation de l'enregistrement audio. Les données seront ainsi écrites dans `audioBuffer` de taille `buffSize` et la condition `nextBufferIsReady` est utilisée pour prévenir qu'un nouveau buffer rempli est disponible.

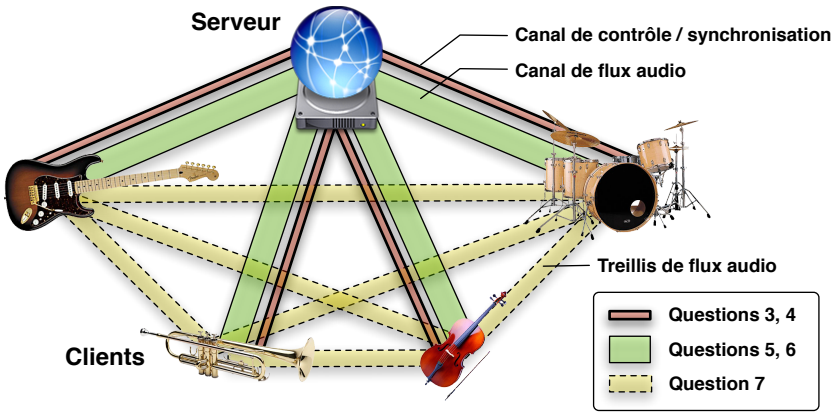

Fig. 1: Modélisation du client-serveur de jam session

`float *receiveAudioBuffer(int socket);`
Fonction permettant de recevoir un buffer audio à travers un `socket`. L'attente sur réception est bloquante.
`int sendAudioBuffer(int socket, float *audioBuffer, float tick);`
Fonction permettant d'envoyer un `audioBuffer` correspondant au temps `tick` à travers un `socket`.
`void enqueueAndPlayBuffer(float *audioBuffer);`
Fonction permettant d'ajouter un `audioBuffer` en queue du buffer de lecture pour l'écouter
Remarque : L'intégralité de l'exercice peut être réalisé dans le(s) langage(s) de votre choix (C, OCaml, Java, Visual Basic, ...).

Question 1. Citer les différents problèmes que pourrait rencontrer un tel système. Quels sont les propriétés requises pour assurer son bon fonctionnement ?

Réponse.

- Problème de *latence* pour le jeu temps réel.
- Problème de *synchronisation* des paquets audio
- Problème de *centralisation / surcharge* à cause de la complexité et la taille des données échangées.
- Propriété de *vivacité* (en termes d'échange de paquet). Les paquets audio doivent arriver le plus vite possible.
- Propriété de *sûreté* (en termes d'échange de paquet). Les paquets audio doivent arriver de tous les clients avant un certain temps sinon le mélange ne pourra pas être assuré.
- Nécessité d'un modèle *consommateur / multi-producteurs* au niveau serveur pour gérer les paquets audio.

Question 2. D'après les fonctionnalités décrites dans l'introduction de l'exercice, définir un protocole de communication entre les clients et serveurs en le divisant entre messages de *contrôle*, de *synchronisation* et de *données.* Pensez à bien spécifier la *directionnalité* éventuelle des messages (client vers serveur ou serveur vers client).

Réponse.
Messages de contrôle
(S => C) EMPTY_SESSION
(S => C) CURRENT_SESSION *style tempo nbMus*
(S <= C) SET_OPTIONS *style tempo*
(S => C) ACK_OPTS
(S => C) FULL_SESSION
Messages de synchronisation
(S => C) NEW_CLIENT *addr*
(S => C) CLIENT_LEFT *addr*
(S => C) AUDIO_PORT *port*
(S => C) AUDIO_OK
(S => C) AUDIO_KO
(S <=> C) AUDIO_SYNC *tick clock*
(S <=> C) AUDIO_ACK
Message de données
(S <= C) AUDIO_CHUNK *buff tick*
(S => C) AUDIO_MIX *buff*

Question 3. Rédiger un serveur permettant de gérer les messages de *contrôle* et *synchronisation* d'un tel système. On ne s'inquiètera pas dans cette première version des problèmes de transfert et synchronisation des flux audio, cependant on devra tout de même mettre en place le second canal de communication pour l'audio. Pour se faire, une fois que la connexion est établie avec le serveur, celui-ci envoie un second numéro de port sur lequel il attendra que le client effectue une seconde connexion pour établir le canal de transfert audio.

Réponse.
On réutilise la classe `echoServerPool` (définie lors du TD.4), en considérant l'ajout des variables *style, tempo* et *audioSocket* (type ServerSocket) à la classe principale. On conserve tout le code en complétant uniquement la méthode `run()` du thread client (`clientThread`) ...

```
public void run() {
 String command, style;
 int nbMusicians, tempo, connected;
 ...
   connected = 0;
   synchronized (server) { nbMusicians = server.newConnect(outchan); }
   if (nbMusicians > MAX_MUSICIANS) {
    outchan.writechars(``FULL_SESSION'');
    server.clientLeft();
    socket.close();
    } else {
     if (nbMusicians == 0) {
      outchan.writechars(``EMPTY_SESSION'');
      command = inchan.readline();
      sscanf(command, ``SET_OPTS %s %f'', style, &tempo);
      server.setStyle(style); server.setTempo(tempo);
     } else {
      outchan.writechars(sprintf(``CURRENT_SESSION %s %f %d'', server.getStyle(), server.getTempo(),
server.getNbConnected());
     }
     outchan.writechars(sprintf(``AUDIO_PORT %d'', server.getAudioPort()));
     ServerSocket audioSock = server.getAudioSocket();
     try {
      Socket clientAudio = audioSock.accept();
      outchan.writechars(``AUDIO_OK''); connected = 1;
     } catch (Exception e) { outchan.writechars(``AUDIO_KO''); }
    if (connected)
     while (true) { // Boucle d'interaction (cf.  Q.6) }
```

Question 4. Rédiger un client correspondant au protocole défini en question 2 et permettant d'interagir avec le serveur de la question 3. Dans cette question on ne s'occupera toujours pas de gérer les flux audio.

Réponse.
On réutilise la classe `echoClient` (définie lors du TD.4), on conserve tout le code en changeant uniquement

```
Socket sock = new Socket(adresse,port);
BufferedReader inchan = new BufferedReader(new InputStreamReader(sock.getInputStream()));
DataOutputStream outchan = new DataOutputStream(sock.getOutputStream());
String answer = inchan.readLine();
if (!strcmp(answer, ``FULL_SESSION'')) {
 if (strcmp(answer, ``EMPTY_SESSION''))
  outchan.writechars(``SET_OPTS %s %f'', style, tempo);
 else
  sscanf(answer, ``CURRENT_SESSION %s %f %d'', style, &tempo, &nbMusicians);
 answer = inchan.readLine();
 sscanf(answer, ``AUDIO_PORT %d'', &audioPort);
 audioSocket = new Socket(adresse, audioPort);
 answer = inchan.readLine();
 if (strcmp(answer, ``AUDIO_OK''))
  while (true)
   { // Boucle d'interaction (cf.  Question 5) }
}
```

Le problème majeur d'un tel système est lié à la latence actuelle des connexions réseaux. En effet, la latence de perception de l'oreille humaine étant environ de 30ms, il est impossible de réaliser une synchronisation temps réelle entre les musiciens. Pour palier à ce problème, l'astuce consiste à forcer une forme de "*latence synchronisée*". L'idée est en fait de décaler les flux audios

de tous les clients d'une mesure (4 temps). Ainsi chaque client joue en entendant le flux audio de tous les autres musiciens décalé de 4 temps.

Question 5. Rédiger la fonction au niveau client permettant d'envoyer des paquets audio avec un timestamp *relatifs*. L'idée ici est d'envoyer avec l'information audio une indication de *tick* qui indique la position de cette information dans la mesure courante (chaque tick correspond à un temps). On considèrera que le premier buffer écrit par la fonction correspond au tick 0 et qu'une mesure correspond à 4 ticks.

Remarques

- Le tempo est défini en *Beats Per Minute* (BPM), ce qui signifie qu'un tempo de 60BPM implique 1 temps par seconde.
- On utilise une fréquence d'échantillonnage de 44100 Hz, ce qui signifie qu'un buffer (vecteur) de 44100 éléments correspond à 1 seconde d'audio.

Réponse.

On crée une classe clientAudioThread, qu'on crée et qu'on lance au tout début de la boucle d'interaction (question 4). La methode run() du thread est définie par

```
class clientAudioThread extends Thread {
 Socket audioSocket;
 Vector<double> audioBuffer;
 int buffSize, samplingFreq;
 float currentTick, tempo;

 clientAudioThread(Socket s, int bSize, float temp, int sFreq) {
  audioSocket = s;
  buffSize = bSize; tempo = temp; samplinfFreq = sFreq;
  audioBuffer = new Vector<double>(buffSize);
 }

 public void run() {
  currentTick = 0;
  initAudioRecording(audioBuffer, buffSize, this);
  while (true) {
   synchronized (this) {
    try { this.wait(); } catch(InterruptedException e) {e.printStackTrace();}
    currentTick += (buffSize / sFreq) * (tempo / 60.0);
    sendAudioBuffer(audioSocket, audioBuffer, currentTick);
   }
  }
 }
}
```

Question 6. Au niveau serveur, l'interaction est plus complexe. Le serveur doit récupèrer les différents morceaux d'audio de chaque client, les synchroniser (en tenant compte des ticks de chaque client) puis mélanger les chunks en un seul (en omettant à chaque fois le client auquel est destiné l'envoi !) qu'il pourra ensuite envoyer aux clients un par un.

Réponse.

Plutôt que de créer une classe spécifique au problème, on peut tout simplement gérer les arrivées de flux audio dans la boucle d'interaction de chaque thread client (cf. question 3). L'idée est de considérer que les threads client s'occupent chacun de maintenir un vecteur de buffers courant. Chaque thread conserve également une liste de ticks (Vector<Double>) permettant d'identifier le timing de chaque buffer du vecteur. Après chaque arrivée d'un flux audio, si un thread client s'aperçoit que suffisamment de buffers sont arrivés chez tous les clients (information maintenue par le *currentTick* de chaque thread), il active la fonction globale *mixAndSend()* qui envoie le flux audio mélangé à chaque client. Cette fonction va récupérer tous les buffers des clients (et les retirer de leurs vecteurs), du tick courant au tick maximum commun pour pouvoir effectuer le mélange.

Le principal problème de l'approche en l'état actuel provient de l'architecture globale du système. En effet, on peut observer que :

1. Le flux audio de chaque musicien doit transiter par le serveur, ce qui ajoute une indirection et donc multiplie grandement la latence
2. Le serveur doit gérer les flux audios de chaque musicien et effectuer toutes les opérations de contrôle et envoi, ce qui peut amener à une surcharge de sa bande passante.

Pour résoudre ces problèmes, on décide de recourir à un "*treillis de connexion*" (lignes pointillées de la Figure 1). L'idée est que les flux audio transitent directement d'un client à l'autre sans passer par le serveur. On doit donc augmenter l'architecture client-serveur pour faire en sorte qu'à chaque connexion réussie, le client courant reçoit l'adresse de tous les musiciens connectés, et effectue une connexion en peer-to-peer à chacun. Les clients devront donc également lancer un mini-serveur permettant d'être à l'écoute en cas de nouvel arrivant.

Question 7. Définir l'architecture nécessaire au niveau des différents clients pour mettre en place ce mécanisme. Rédiger de manière prototypique les différentes structures et fonctions requises (au niveau serveur et client) pour créer le treillis de communication audio lors de l'arrivée d'un nouveau client.

Réponse.
Côté serveir :
On conserve une liste d'adresses (Vecteur) des clients. A chaque nouvelle connexion, au lieu d'ouvrir le second socket (pour l'audio), le serveur envoie la liste des adresses aux clients. On peut ensuite retirer toute gestion audio depuis le serveur principal. On conserve cependant les fonctions de mélange audio identique qu'on utilisera dans chaque client (pour que chacun s'occupe soit-même de mélanger les flux des autres clients. On ajoute cependant la gestion des messages NEW_CLIENT et CLIENT_LEFT au protocole (sous forme d'information broadcast)
NEW_CLIENT : L'adresse du nouveau client est envoyée à tous les clients courants
CLIENT_LEFT : Le départ d'un client est annoncé à tous
Côté client :
On ajoute la même liste d'adresse, et on ajoute également un serveur de gestion de flux audio. Cette fois-ci on ajoute une fonction *connectToAll*, qui permet à un client de se connecter à tous les musiciens présents. Cette fonction doit gérer la connexion réciproque