Network musical jammin'

Projet PC2R - 2015

Pour ce projet, nous allons réaliser une application 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

1 Présentation

Nous considèrerons dans ce projet qu'il n'existe qu'une seule jam par serveur (tous les clients jouent tous ensemble). De plus, sauf en cas d'implémentation de l'extension P2P (cf. section 5.1), nous considèrerons également que la communication est *centralisée*, ie. le serveur gère toutes les interactions. 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.

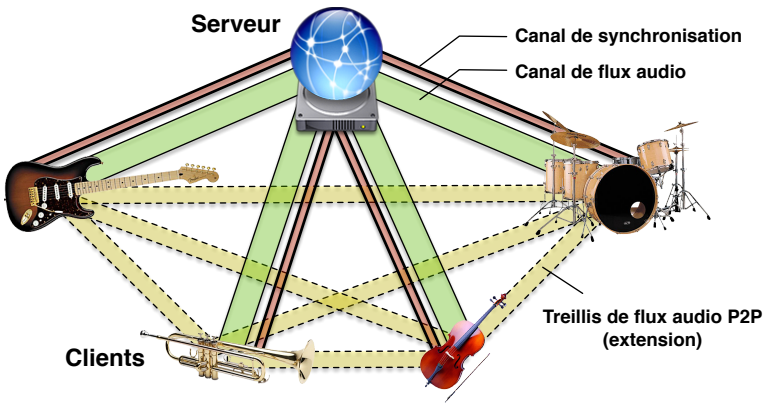

Fig. 1: Modélisation du client-serveur de jam session

Dans ce projet, il faudra être très attentif aux problème de *latence* pour le jeu temps réel et de *synchronisation* des paquets audio (on verra que la seule solution consiste à décaler le jeu des autres musiciens d'un nombre fixe de temps et de resynchroniser le flux chez chaque musicien). De plus, les problèmes de *centralisation / surcharge* à cause de la complexité et la taille des données échangées nécessiteront un modèle *consommateur / multi-producteurs* au niveau serveur pour gérer les paquets audio, ainsi qu'un système de *pool de threads* pour assurer la vivacité.

2 Principes du traitement audio

Le but du projet étant de recevoir de l'audio (depuis l'entrée micro ou autre entrée du client), puis de le mélanger pour l'envoyer aux différents clients, nous rappelons ici quelques concepts de base de l'information audio.

1. L'information audio est stockée sous forme d'un vecteur de nombres flottants (généralement avec des valeurs dans l'intervalle [-1,1]).
2. Le rapport entre la durée d'un son et le nombre de valeurs dans ce vecteur dépend d'une *fréquence d'échantillonnage*. Par défaut, les systèmes sont paramétrés sur une fréquence de 44100 Hz. Cela signifie tout simplement qu'une seconde de son produira 44100 valeurs dans le vecteur. **Cependant les buffers envoyés ne doivent pas forcément faire 44100 valeurs, on préfère la *vivacité* en envoyant des buffers plus "classiques" de 4096 ou 8192 valeurs.**
3. "Mélanger" différents sons consiste simplement à additionner les différents vecteurs. Si le vecteur résultant de l'addition est envoyé en lecture à la carte son, tous les sons seront joués en même temps. La seule précaution lors de l'addition est de s'assurer que les valeurs restent dans l'intervalle [-1,1] (en cas de dépassement, on pourra tout simplement diviser toutes les valeurs par le maximum).
4. Le *tempo* d'une chanson est définie en BPM (Beats Per Minutes), qui donne le nombre de temps par minutes. Par exemple un tempo de 120 BPM signifie qu'on tapera deux fois du pied par seconde.
5. La problématique de *synchronisation* dans ce projet est de bien découpler la notion de temps physique (en secondes) et de temps logique musical (par rapport au tempo de la chanson). Ainsi chaque personne joue par rapport au temps logique musical (les buffers envoyés contiennent donc un *tick* de début, pouvant être interprété comme "je jouais cette partie en tapant pour la 3ème fois du pied"). Le rôle du serveur est donc d'interpréter ce temps logique, pour mélanger les buffers des clients *correspondant au même temps logique*.
6. Le principe de *décalage* (ou latence forcée) est de s'assurer que tout le monde puisse jouer "en rhythme" (pour gérer la latence du réseau) en décalant l'audio d'un certain nombre de temps logique. Pour simplifier, si on décide de décaler de 4 temps cela signifie que si je suis en train de jouer en tapant du pied pour la 7ème fois, j'entends ce que les autres jouaient en tapant du pied pour la 3ème fois.

3 Travail à réaliser

1. Programmer un serveur et un client *dans deux langages différents*, communiquant par le protocole décrit dans ce document. Le serveur gérera la connexion des musiciens, le démarrage de la jam, les connexions possibles des musicien en cours de jam, le choix du tempo (par le créateur de session), la transmission du flux audio de chaque musicien, la diffusion des flux de chaque musicien à l'ensemble des musiciens. Le client gérera la demande de connexion au serveur, l'attente du début de jam, l'envoi de l'audio local au fur et à mesure et la lecture du flux entrant (plus éventuellement avec les extensions).
2. Rédiger un petit rapport de 4 à 8 pages, avec une limite de deux captures d'écran et des marges et tailles de fonte raisonnables.

4 Protocole

Le serveur écoute sur le port 2013 en TCP. Les deux parties parlent un protocole texte qui prend la forme d'une succession de commande. Chaque commande est composée d'une suite de chaînes terminées par des caractères '/'. La dernière chaîne est suivie d'un retour chariot Unix `\n`. La première chaîne est le nom de la commande, les autres ses arguments. Chaque nom de commande n'a qu'un seul rôle dans le protocole (le même nom de commande ne peut signier deux choses différentes à deux moments différents). La suite "`\/`" dans un argument est utilisé pour échapper le caractère '/', et "`\\`" le caractère '\'.

Conseil: tirez parti des possibilités des langages utilisés pour rendre automatique et sûre la lecture et l'interprétation des commandes.

4.1 Connexion/Deconnexion

Lorsque le client se connecte au serveur il envoie sans attendre la commande `CONNECT/user/` suivie du nom du musicien précisé par l'utilisateur. Le serveur répond par une commande `WELCOME` suivie du nom du musicien (qui peut être un peu différent dans le cas où plusieurs musiciens auraient demandé le même nom). Après réception de ce message, on devra tout de mettre en place le second canal de communication pour l'audio. Pour ce faire, le serveur envoie immédiatement après le message `WELCOME/user/` un message `AUDIO_PORT/`*`port/`* contenant un second numéro de port sur lequel il attendra que le client effectue une seconde connexion pour établir le canal de transfert audio. Après établissement du canal, le serveur envoi un message `AUDIO_OK` *sur le canal de synchronisation (principal)* du nouveau musicien ainsi qu'une commande `CONNECTED` suivie du nom du musicien à tous les musiciens (y compris lui-même).

```
CONNECT/user/
(C->S) Nouvelle connexion de "user"
WELCOME/user/
(S->C) Signifie au musicien qui a demandé la connexion que celle-ci est acceptée sous le nom "user"
AUDIO_PORT
(S->C) Signifie au musicien que le serveur attend une connexion sur le port audio
AUDIO_OK/port/
(S->C) Signifie que le canal audio est établi
CONNECTED/user/
(S->C) Signifie à tous les clients la connexion de "user"
```

Il peut avoir des déconnexions d'utilisateur en cours de partie: soit si le musicien l'indique explicitement, soit en cas de femeture de la communication entre un client et un serveur.

```
EXIT/user/
(C->S) Déconnexion de "user"
EXITED/user/
(S->C) Signifie à tous les clients le départ de "user"
```

4.2 Gestion des paramètres de Jam

Les jams se jouent de 2 à MAX musiciens, si la jam est vide, le serveur enverra un message `EMPTY_SESSION` auquel cas le premier musicien peut régler les paramètres de jam en envoyant un message `SET_OPTIONS` contenant le style (chaîne de caractères) et le tempo, auquel le serveur répond par un `ACK_OPTS`. Si la session n'est pas vide, le nouvel arrivant reçoit le message `CURRENT_SESSION` contenant le style, tempo et nombre de musiciens. Si la session est pleine, le nouvel arrivant reçoit un message `FULL_SESSION`, auquel cas il peut soit quitter proprement (message `EXIT/user/`), soit se mettre en mode spectateur (cf. Extensions) jusqu'à ce qu'un musicien quitte la jam.

```
EMPTY_SESSION
(S->C) Signale au client que la session est vide
CURRENT_SESSION/style/tempo/nbMus/
(S->C) Signale au client les paramètres de la jam
SET_OPTIONS/style/tempo/
(S<-C) Message du client pour mettre les paramètres (en cas de session vide)
ACK_OPTS
(S->C) Signale la bonne réception des paramètres
FULL_SESSION
(S->C) Signale au client que la session est pleine
```

4.3 Gestion des flux audios

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. L'idée ici est d'envoyer au serveur un message *sur le canal audio* de type `AUDIO_CHUNK/`*`tick/buff/`* contenant le buffer audio et une indication de *tick* (flottant) qui indique la position de cette information dans la mesure courante (chaque tick correspond à un temps). Il s'agit donc d'une forme de *timestamp relatif*. On considèrera que le premier buffer écrit par la fonction correspond au tick 0 et qu'une mesure correspond à 4 ticks. Le serveur répondra sur le canal de contrôle par un message `AUDIO_OK` ou `AUDIO_KO` en cas de problème.

AUDIO_CHUNK*/tick/buff/*
(C->S) Envoi d'un buffer audio au serveur
AUDIO_OK
(S->C) Bonne réception du buffer
AUDIO_KO
(S->C) Problème de réception

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 sous forme d'un message AUDIO_MIX*/buff/*. Le client répondra par un message AUDIO_ACK.

AUDIO_MIX*/buff/*
(S->C) Buffer contenant le mélange global des autres musiciens
AUDIO_ACK
(C->S) Bonne réception du buffer audio

A noter que deux cas particuliers doivent être gérés. Tout d'abord lorsqu'un nouvel arrivant veut jouer dans la jam, celui-ci doit se synchroniser. En plus des paramètres, le serveur envoie alors un message AUDIO_SYNC/tick/ indiquant la dernière valeur de tick envoyée. D'autre part, si un musicien à des problèmes de connexion, on ne veut pas que toute la jam soit bloquée. Ceci devra être géré par un mécanisme de timeout, permettant d'omettre un musicien du flux audio global si il met trop de temps à repondre (et envoi d'un message AUDIO_KO à ce musicien). Au bout de 8 messages AUDIO_KO successifs, le serveur force la déconnexion du musicien fautif.

Remarques

- Notez bien que le serveur retire *volontairement* le musicien de son propre AUDIO_MIX car sinon celui-ci s'entendrait lui-même décalé de quatre temps, il faudra donc que le client mélange également le flux global au flux local avant de l'envoyer à la carte son.
- 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.

4.4 Fin de jeu

Lorsqu'un musicien se déconnecte, tous les autres sont prévenus. Si c'était le musicien originel (celui ayant mis en place les paramètres de la jam), le serveur arrête la jam courante et propose au musicien suivant de changer les paramètres de la jam, puis celle-ci recommence avec la nouvelle configuration. Si plus aucun musicien n'est dans la jam, les paramètres sont effacés et le serveur se remet en attente.

4.5 Lancement du serveur et des clients

Le serveur prendra en paramètre le nombre de musiciens maximal, la durée du timeout (en milisecondes) après laquelle un musicien n'est pas considéré dans le flux audio:
`./serveur -max n -timeout t -port p`
par défaut : n vaut 4, t 1000 milisecondes, port 2015.
Le client prendra en paramètre le numéro de port pour se connecter au serveur et le nom du musicien:
`./client -port p -user name`
par défautl le port vaut 2015 et name pc2r.

5 Extensions

Ces extensions sont à réaliser une fois que le noyau du jeu fonctionne. Elles setont prises en compte pour l'évaluation de ce devoir de programmation.

5.1 Gestion peer-to-peer

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.

5.2 Discussion instantanée

Le client doit afficher une petite zone de discussion instantanée façon IRC. Lorsque le musicien a écrit son message et l'envoie, le client envoie une commande `TALK` au serveur suivie du texte : `TALK/texte/`. Le serveur transmet alors à tous les clients la commande `LISTEN` suivie du nom du musicien puis du message : `LISTEN/musicien/texte/`. Chaque client doit alors afficher le message dans sa console de discussion.

Notes :

- Toutes ces communications se feront par le canal de contrôle

- le serveur peut envoyer lui-même des messages (par exemple en utilisant le nom "(broadcast)"). Cela peut être utile pour ajouter des décomptes ou des commentaires.

5.3 Comptes utilisateurs

En plus du mode anonyme, on veut permettre à un musicien de réserver son nom, en protégeant son utilisation par un mot de passe. Pour ceci, on autorise deux commandes de connexion supplémentaires :

- `REGISTER/n/p/` se connecte sous le nom n et réserve ce nom pour de futures connexions, en le protégeant par le mot de passe p. Si l'utilisateur existe déjà dans la base, le client reçoit une commande `ACCESSDENIED` et la connexion est terminée.

- `LOGIN/n/p/` se connecte sous le nom n, qui doit être enregistré dans la base d'utilisateurs et dont le mot de passe doit être p. Si l'utilisateur n'existe pas où si le mot de passe ne correspond pas, le client reçoit une commande `ACCESSDENIED` et la connexion est terminée.

Notes : le choix d'implantation de la base d'utilisateurs est laissée libre (fichier texte, base SQLite, donnée sérialisée, etc.). En option, vous pouvez établir un protocole plus sûr que la transmission du mot de passe en clair.

5.4 Serveur HTTP d'enregistrement

On veut que le serveur enregistre périodiquement les flux audios pour les conserver sous forme de fichiers WAV (on considèrera le début d'une jam comme le moment ou 2 musiciens sont connectés et la fin quand il n'en reste plus qu'un ou zéro). En accédant au port 2092 de la machine où est lancé le serveur depuis un navigateur Web, on obtient alors une page HTML indiquant pour chaque musicien inscrit dans la base de données son ensemble d'enregistrements. Il est alors possible d'écouter les différentes jams réalisées par chaque musicien.

Indication : pour vous éviter la lecture de la spécification du protocole HTTP, interrogez quelques sites Web directement avec la commande telnet pour vous donner une première idée de ce que doit renvoyer votre serveur.

6 Mode Spectateur

On rajoute une commande de connexion `SPECTATOR`. Un spectateur n'intervient pas dans la jam mais peut entendre tous les musiciens actifs. On considère que le spectateur ne pourra entendre la jam qu'à partir de sa connexion (pas d'envoi des enregistrements passés).