Thomas Guzik & Adam Morel

Compte-rendu

Introduction

Ce programme permet de créer un système client-seveur de streaming musical. Le client envoie le nom de fichier avec des filtres, et le serveur lui envoie le son en retour. Le serveur ne sait jouer que des fichiers wav. À cause des limitations techniques du à la bibliothèque audio.c, le serveur ne peut gérer que les fichiers wav sur 8bit/16bit sur 1 ou 2 channels (les fichiers 32bit provoquant une erreur lors de leur ouverture).
Le serveur gère le multi-client en utilisant un fork dedié pour chaque client. Du coté client on peut demander au serveur de jouer un fichier wav avec 0,1 ou plusieurs filtres.
Les differents filtres disponiles sont `MONO` `VOLUME` et `ECHO`.
Pour commencer à utilser le serveur, il faut d'abord commencer par créer les fichiers executables. Ceux ci sont crée par la commande make:

```
make
```

Par la suite, on peut lancer l'executable du serveur par la commande :

```
./audioserver
```

Une fois le serveur mis en marche, on peut commencer à lui envoyer des requêtes. Pour executer un client on utilise cette commande suivi du nom des filtres possibles

```
padsp ./audioclient nomdudomaine nomdufichier.wav FILTRE1 FILTRE2 FILTRE3
```

Par exemple :

```
padsp ./audioclient localhost test30s.wav ECHO MONO
```

Implémentations techniques

Un aperçu du code est résumé par le schéma sur la page 3

Implémentation des sockets

La gestion des sockets se fait via socketlvl2.h socketlvl2.c Cette mini-bibliothèque implémente des fonctions permettant une gestion plus aisée de socket.h

Protocole d'envoi

Le protocole d'envoi est le suivant:
Le client envoie le nom du fichier avec un filtre. La representation en mémoire du nom avec le filtre est un tableau de char avec comme premier char le filtre suivi du nom. Le filtre est représenté par un bit, MONO = 0b001, VOLUME = 0b010, ECHO = 0b100. Biensûr les trois bits peuvent être en meme temps. Le client envoie donc ce tableau au serveur.
Si le client ne reçoit aucune réponse du serveur, l'erreur connection timeout est soulevé.
Le serveur reçoit les données. Si le serveur n'a pas le fichier demandé, il enverra un message d'erreur 404. Si le serveur est surchargé, il enverra le message d'erreur 503. Enfin si le filtre demandé n'est pas possible pour le fichier demandé (par exemple on demande du mono pour un fichier de 1channel), alors le serveur enverra l'erreur 405. Après cette vérification des erreurs, le serveur peut se lancer dans l'envoi des données. Le serveur enverra tout le fichier jusqu'à atteindre sa fin. Il enverra une socket vide pour signaler la fin de transmission.

Implementation de l'envoi en temps réel

Le serveur a besoin d'envoyer des paquets de données au bon moment afin que le son ne soit joué ni trop vite, ni trop lent. La première implémentation était d'envoyer les données et de faire ensuite dormir notre serveur à l'aide d'un sleep. Malheureusement, cette implémentation n'est pas efficace, la fonction sleep ne permettant pas de préciser à quel moment seront envoyer les données exactement. Nous utilisons un timerfd, celui ci crée un file descriptor vers un timer dont on a précisé la durée des ticks. (Ce timer tick toutes les n nanosecondes) À chaque tick, on lira donc les données et on enverra les data. Pour recevoir le tick du timer, on passe par la fonction read. Le read est bloquant tant que le timer n'a pasické

```
while(read(timerfd, &overrun, sizeof(overrun)) > 0) {
        // Si on le nombre de bytes lu == 0, alors on a fini de lire le fichier
        if((bytes_read = read(s->read,buf, bytes_to_read)) == 0) {
                break;
```

```
        }
        else {
                sendto(sockfd, buf, bytes_read, 0,(struct sockaddr*)&client,clientlen);
        }
}
```

Timerfd utilise la bibliothèque timerfd.h

Implémentation du multi-client

L'implémentation du multi-client demande qu'une partie du programme s'occupe de la reception des requêtes client tandis qu'une autre partie s'occupe d'envoyer les données au client. Pour cela, nous utiliserons un système de fork. À chaque nouvelle requête, nous crérons un fork. Le processus père s'occupera toujours de recevoir les requêtes tandis que le processus fils enverra les données à au client. Lorsque le processus fils se termine, il passe en état zombie et envoie un signal à son processus père. Pour pouvoir éliminer ce zombie, le père doit récupérer le signal de son fils mort. Or le père a un énorme problème pour faire cela. Il est bloqué à attendre une socket client dans la fonction recvfrom(). Pour que le père puisse récupérer le signal de son fils, il doit pouvoir sortir de la fonction recvfrom(). recvfrom() doit donc être non bloquant. Pour ce faire, nous utilisons la fonction select() permettant de mettre un timeout au fonction d'entrée sortie. Un timeout juste semble être d'une seconde. Ansi chaque seconde, le père vérifie si un de ses fils est mort.
Le serveur peut également supporter qu'un certain nombre de client à la fois. Ici ce nombre est fixé à 3 par `MAX_PID`.

Implémentation des filtres

Les filtres ont été implémenté dans `lecteur.h` et `lecteur.c`

Filtre Mono

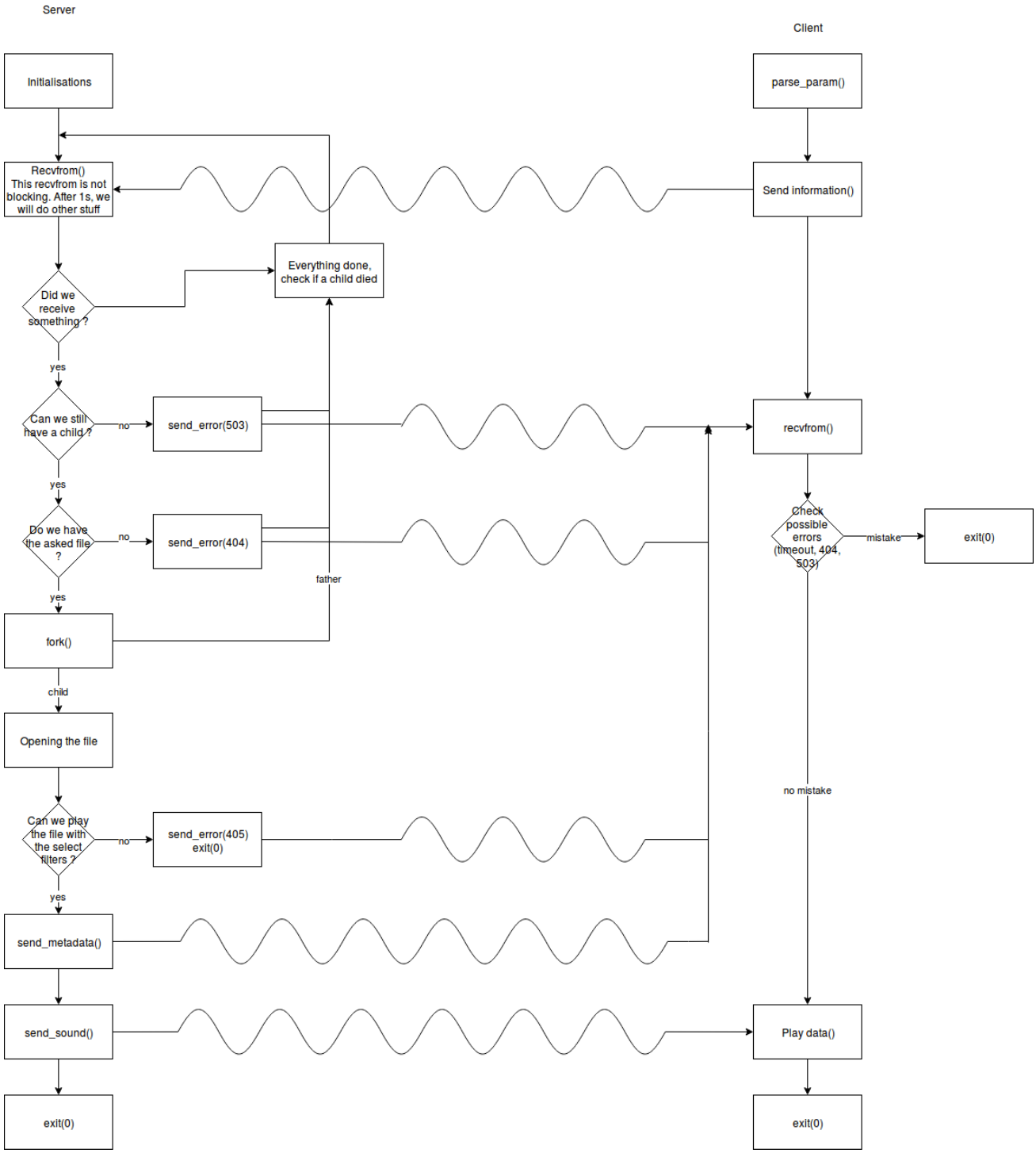
Le filtre mono permet de passer d'un fichier ayant 2 channel à un fichier n'ayant plus que 1 channel.
L'implémentation est la suivante :
Un coup on joue un échantillon gauche, un coup on joue l'échantillon droite. Faire la moyenne des 2 échantillons produit un bruit de fond désagréable.
Le filtre mono est uniquement disponible pour les fichiers contenant 2 channels. Il est impossible de le faire pour les fichiers contenant seulement 1 channel.

Filtre Volume

Le filtre volume diminue le son. La prise en compte d'une valeur n'a pas eu le temps d'etre implémenté (bien que la fonction volume permet de choisir son niveau) Ansi dès qu'on appelle la fonction, le volume sera réduit de 20%.
Le filtre volume ne peut pas être disponible pour les fichiers 8bit, car l'augmentation d'un signal produit des distorsions trop forte.

Filtre Echo

Le filtre echo produit un echo. Pour cela on utilise un buffer (bufecho) qui sauvegarde les précédents échantillons et on les additionne à l'échantillon actuel.
Ce filtre est disponible uniquement pour les fichiers 16bit.


Server
Client
Initialisations
parse_param()
Recvfrom()
This recvfrom is not blocking. After 1s, we will do other stuff
Send information()
Everything done, check if a child died
Did we receive something ?
yes
Can we still have a child ?
no
send_error(503)
recvfrom()
yes
Do we have the asked file ?
no
send_error(404)
Check possible errors (timeout, 404, 503)
mistake
exit(0)
father
yes
fork()
child
Opening the file
no mistake
Can we play the file with the select filters ?
no
send_error(405)
exit(0)
yes
send_metadata()
send_sound()
Play data()
exit(0)
exit(0)