Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2013-2014**

Examen réparti du 05 novembre 2013

Exercice : Simulation d'un rayon dans un supermarché (Fair threads)

On veut simuler le fonctionnement d'un rayon dans un supermaché, par exemple un rayon de poissonerie avec des clients et un serveur :

- Chaque client arrive d'une manière aléatoire (voir le macro `RAND(n)` ci-dessous), prend un ticket numéroté (voir la procédure `prendre_ticket()`) et attend l'affichage de son numéro pour être servi (voir `numero_affiche` ci-dessous). Si le numéro de son ticket ne correspond pas à celui de l'affichage, il peut prendre "aléatoirement" l'un des 3 choix possibles :
  1. Il attend "passivement" un nombre n (aléatoire) de tours pour être servi. Et au bout de ces n tours non fructueux, il reconsidère son choix.
  2. Il décide de ne pas attendre, quitte la file en gardant son `ticket`. Si à son retour, le numéro de son ticket est dépassé, il est obligé de reprendre un nouveau ticket et reconsidère son choix.
  3. Il abandonne tout simplement le rayon en gardant son ticket.

  Si son numéro est affiché, il se présente au serveur (en affectant la variable `present` avec le numéro de son ticket), quitte la file, part avec le serveur et ne revient plus. Bien sûr, pendant ce temps, les autres clients continuent normalement leur activité.
- Le serveur
  1. affiche un nouveau numéro (en incrémentant la variable `numero_affiche`),
  2. laisse au plus 3 instants au client portant ce numéro de se présenter (voir la variable `present`),
  3. au bout de ces 3 instants, il recommence son étape 1.
  4. si le client est présent (la variable `present` porte le numéro de son `ticket`), le serveur lui sert pendant un certain temps (aléatoire). Bien sûr, pendant ce temps, les autres clients continuent normalement leur activité.
  5. il reviend et recommence l'étape 1.

  Si aucun client n'est présent (`numero_ticket <= numero_affichage`), il attend "passivement" la présence (voir `present`) d'un client.

Question 1 Dans l'annexe ci-dessous, ajouter les contantes et/ou variables que vous jugez nécessaires pour cette simulation.

Question 2 Compléter la procédure `client()` pour le client qui doit afficher tous ses faits et gestes.

Question 3 Compléter la procédure `serveur()` pour le serveur qui doit afficher tous ses faits et gestes.

*/***************************** DEBUT ANNEXE ************************************/*

```c
#include "fthread.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "traceinstantsf.c"

#define MAX_CLIENTS     10
#define MAX_SERVEURS    2
```

```c
/*
  RAND(n) : rend un nombre aleatoire compris
            entre 0 et n, bornes comprises.
*/
#define RAND(n) ((int)((rand() * 1.0 / RAND_MAX) * n))

int             numero_ticket  = 1,
                numero_affiche = 0,
                present        = 0;
ft_scheduler_t rayon;
ft_thread_t    serveurs[MAX_SERVEURS], clients[MAX_CLIENTS];
```

```c
/* ?????????? */

int prendre_ticket () {
 return numero_ticket++;
}

void client (void *args) {
 int ticket = prendre_ticket();
 ft_scheduler_t rayon =
    ft_thread_scheduler();
 int numero_client = (int)args;

  /* ?????????? */
}

void serveur (void *args) {
 int i, tour = (int)args;

  /* ?????????? */
}
```

```c
int main (void) {
 int         i;
 ft_thread_t thread_array[MAX_CLIENTS];

 rayon             = ft_scheduler_create();
 reveiller_client  = ft_event_create(rayon);
 reveiller_serveur = ft_event_create(rayon);

 ft_thread_create(rayon, traceinstants, NULL, (void *)50);
 ft_thread_create(rayon, serveur, NULL, (void *)100);
 ft_scheduler_start(rayon);

 for (i = 0; i < MAX_CLIENTS; i++) {
  sleep(RAND(5));
  thread_array[i] =
   ft_thread_create(rayon, client, NULL, (void *)i);
 }
 ft_exit(); /* bloquant : permettant que tout le monde
 return 0;  /*   ait le temps de tout faire */
}
```

*/***** Un extrait de l'execution du programme*

```
$ Exo1
>>>>>>>>>>> instant 0 :
Le serveur affiche le numero 1.
>>>>>>>>>>> instant 1 :
...
>>>>>>>>>>> instant 4 :
Plus de client : le serveur attend.
>>>>>>>>>>> instant 5 :
...
Un client arrive ...
Le serveur affiche le numero 2.
Le serveur sert le client ayant le ticket numero 2.
Le client nom_1 ayant le ticket numero 2 se presente au serveur.
...
Le serveur revient pour servir un nouveau client.
Le serveur affiche le numero 3.
...
Le client nom_2 ayant le ticket numero 3 se presente au serveur.
Le serveur sert le client ayant le ticket numero 3.
Le client nom_3 ayant le ticket numero 4 quitte la file.
...
Le serveur revient pour servir un nouveau client.
Plus de client : le serveur attend.
Le client nom_3 ayant le ticket numero 4 reviend.
Il arrive trop tard et doit reprendre un nouveau ticket.
Son nouveau ticket porte le numero 7.
Le client nom_3 ayant le ticket numero 7 se presente au serveur.
...
Le client nom_8 ayant le ticket numero 14 se presente au serveur.
Le serveur sert le client ayant le ticket numero 14.
Le serveur revient pour servir un nouveau client.
Le serveur affiche le numero 15.
Plus de client : le serveur attend.
...
```

*/************************** FIN ANNEXE ************************************/*

Problème : Jam session en client-serveur

Dans ce problème, 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

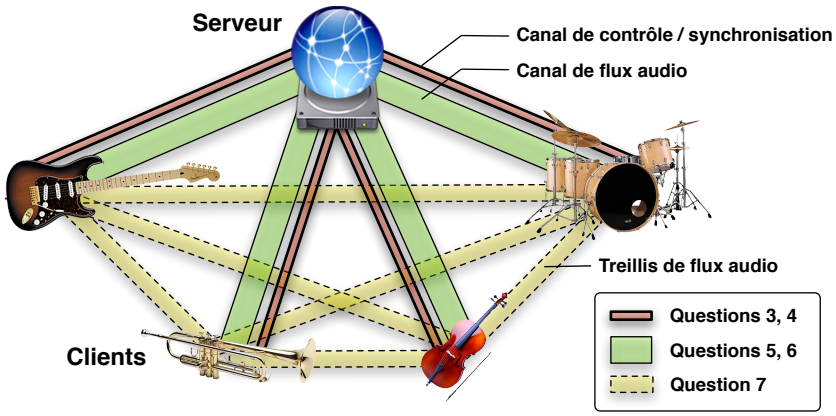

Fig. 1: Modélisation du client-serveur de jam session

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.

```
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 entre C, OCaml et Java. Si vous n'utilisez pas C, donnez les interfaces des fonctions ou méthodes d'Audio dans le langage utilisé.

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

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).

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.

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.

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.

Question 6. Au niveau serveur, l'interaction est plus complexe. Le serveur doit récupérer les différents morceaux d'audio de chaque client, les synchroniser (en tenant compte des ticks de chaque client) puis mélanger les morceaux 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.

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 d'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.