BOSSA et le Concert Virtuel Réparti, intégration et paramétrage souple d'une politique d'ordonnancement spécifique pour une application multimédia distribuée

J. Cordry, N. Bouillot, S. Bouzefrane

Laboratoire CEDRIC, CNAM, 292 rue Saint Martin, Paris Cédex 03, France.
julien.cordry@auditeur.cnam.fr,
{ bouillot, samia.bouzefrane}@cnam.fr

Résumé

Le CEDRIC et l'IRCAM mènent depuis 2002 un projet intitulé "concert réparti" qui se propose de coordonner sur un réseau les acteurs d'un concert musical (musiciens, ingénieur du son, auditeurs) en vue de produire un concert "Live". Au niveau de chaque site (musicien) du réseau, principalement deux composants sont actifs : le moteur de son audio (FTS) et le module d'auto-synchronisation des musiciens (nJam), deux modules qui doivent traiter des flux audio en temps réel et les échanger via le réseau. Nous proposons dans cet article d'ordonnancer les processus générés par ces composants en utilisant une technique d'ordonnancement temps réel. Pour cela, nous choisissons d'utiliser BOSSA, une plate-forme qui se greffe sur le noyau Linux en vue d'intégrer de nouveaux ordonnanceurs temps réel.

Plan

Mots Clés

Ordonnancement temps réel, application multimédia, système temps réel.

1. Introduction

Le concert virtuel réparti est le résultat d'une coopération entre les laboratoires de recherche de l'IRCAM[1] et du CEDRIC[2]-CNAM [Bou., 2003; Loc. et al., 2003]. Le but de ce projet est de permettre à des musiciens physiquement éloignés de jouer ensemble en temps réel via un réseau IP multicast. Les musiciens échangent des flux audio de type PCM, permettant une restitution du son haute qualité. Lors de la phase de jeu, chaque musicien diffuse la musique qu'il joue. Il entend sa propre musique mixée avec celle des autres, mais après une latence maintenue constante.

Le prototype du concert réparti fût initialement développé dans un environnement Linux classique (ordonnancement préemptif). Pour une utilisation dans un contexte de laboratoire, les tests d'utilisation se font avec des machines dédiées où aucun autre processus ne charge le système. La même hypothèse ne peut exister dans un contexte réel d'utilisation, c-à-d, le musicien chez lui. Nous devons donc considérer que de toute façon, d'autres processus, éventuellement gourmands en termes de consommation processeur et de bande passante réseau, cohabitent.

Notre objectif est de définir quel ordonnancement(s) s'adapte(nt) aux contraintes du concert réparti. BOSSA est une plate-forme orientée événements qui s'utilise sur le noyau Linux et qui permet d'intégrer de manière souple de nouvelles politiques d'ordonnancement. Notre choix s'est donc porté sur l'utilisation de BOSSA pour les raisons suivantes :

- pour bénéficier d'un ordonnancement «BOSSA», les modifications du code de notre application nécessite un minimum d'insertion de code (trois lignes de code par processus pour l'attacher à un ordonnanceur spécifique). En effet, BOSSA est une réécriture sous forme de règles appliquées automatiquement aux sources du noyau Linux. On aurait pu utiliser un Linux temps réel tel que RTAI[3] mais cela aurait demandé la réécriture complète de l'application vu la structure particulière des tâches temps réel et les librairies particulières à utiliser.

[1] http://www.ircam.fr
[2] http://cedric.cnam.fr
[3] Real-Time Application Interface, développé à Dipartimento di Ingeniera Aerospaziale, Politecnico di Milano du Prof. Paola Mantegazza (http://www.aero.polimi.it/~rtai/applications/)

- Nous pourrons utiliser un ordonnanceur temps réel pour la gestion des processus de notre application multimédia, autre que ceux de Linux (SCHED_FIFO pour une gestion FIFO, SCHED_RR pour une gestion Round Robin).

Dans la section 2, nous décrivons les caractéristiques de notre application multimédia, le concert virtuel réparti. Dans la section 3, nous présentons les caractéristiques de la plate-forme BOSSA et nous montrons comment elle peut intégrer et faire cohabiter de nouvelles politiques d'ordonnancement en utilisant la notion de hiérarchie d'ordonnanceurs. Dans la section 4, nous déterminons les processus de l'application qui doivent être ordonnancés en temps réel en définissant une hiérarchie d'ordonnanceurs. Avant de conclure en section 6, la section 5 présente un certain nombre de courbes de mesure qui montrent bien que les performances de l'application sont meilleures lorsque l'application tourne sous BOSSA.

2. Le concert virtuel réparti

L'IRCAM et le CNAM-CEDRIC mènent un projet depuis 2002 nommé le "concert virtuel réparti" qui se propose de coordonner sur un réseau les acteurs d'un concert musical (musiciens, ingénieur du son, auditeurs) en vue de produire un concert "Live" (voir Figure 1) [Bou., 2003; Loc. et al., 2003].

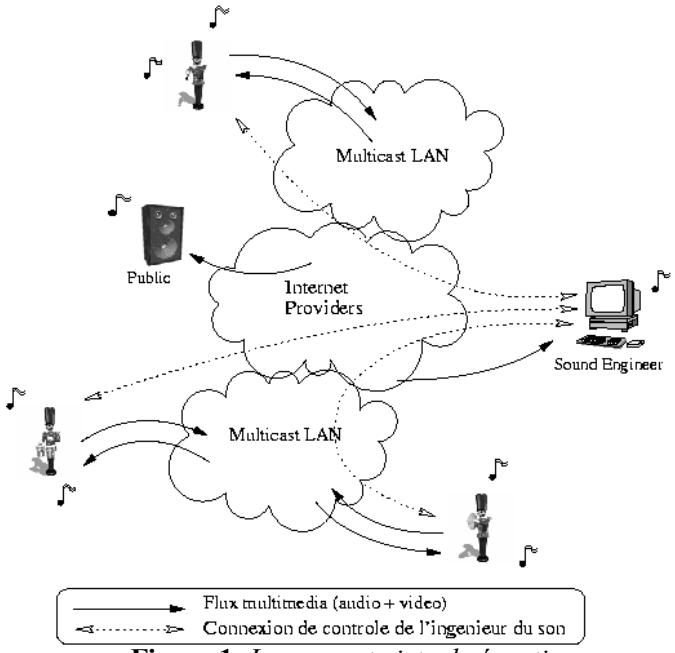


Figure 1. *Le concert virtuel réparti*

Les contraintes applicatives sont les suivantes:

- Les musiciens sont physiquement répartis mais doivent jouer "virtuellement ensemble" en temps réel.
- L'ingénieur du son doit pouvoir effectuer des réglages en temps réel sur les paramètres audio des différentes sources de son. Par exemple, faire une balance, rajouter des effets de réverbération, etc.
- Le public doit pouvoir assister virtuellement au concert, soit à la maison par un mécanisme standard de streaming audio/vidéo, soit en salle avec une installation dédiée.

Nous allons ici nous intéresser uniquement à la partie concernant les musiciens, puisque c'est une partie critique en terme d'interactivité. Des travaux sur la partie ingénieur du son ont déjà été initiés [Loc. et al., 2003] et les travaux sur la diffusion vers le public sont en cours. Notre application utilise *jMax,* un environnement de programmation visuelle pour des applications musicales et multimédia temps réel. *jMax* [Déch., 2000], qui est développé à l'IRCAM, est composé de deux parties: *FTS* pour "faster than sound", un moteur temps réel statistique et une interface graphique qui permet d'ajouter, de supprimer, de paramétrer et de relier

des composants qui vont communiquer par flux d'échantillons audio ou par valeurs discrètes. Les entrées sorties de la carte son, les opérations arithmétiques, les filtres audio-numériques sont des exemples de composants disponibles dans *jMax*. Il possède une interface avec ALSA (pour Advanced Linux Sound Architecture) sous Linux. La Figure 2 nous montre un exemple d'assemblage simple de composants *jMax*. Le composant "osc 440" va générer une sinusoïde avec une fréquence de 440 Hz (correspondant à la note "la") puis envoyer ce signal vers le composant du dessous, comme l'indique le lien entre eux (le sens de transmission est du haut vers le bas). Ce deuxième composant est la sortie audio configurée dans les menus de *jMax*. Cet assemblage va donc jouer la note "la" en continu sur les haut parleurs.

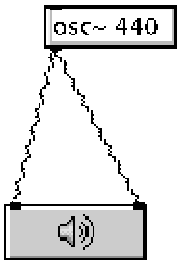


Figure 2. *Un assemblage de composants jMax*

Dans des conditions de type "réalité virtuelle", la qualité du son, le sentiment de présence, ainsi que le synchronisme entre musiciens sont de la première importance. Pour cela, la technologie développée pour le concert réparti utilise du son non compressé (nous utilisons des échantillons PCM à 44100Hz, ce qui correspond à la qualité d'un CD audio du commerce) ainsi que la diffusion en multi-cast à l'aide du protocole RTP [Sch. et al., 1998] pour la communication entre musiciens. Pour le sentiment de présence, lors de nos expérimentations avec des musiciens, nous avons utilisé un logiciel de vidéo-conférence pour permettre aux musiciens de se voir.

L'interaction musicale "classique" (tous les musiciens dans la même salle) est basée sur une notion de perception commune: dans une même salle, les événements sonores et visuels sont perçus instantanément, simultanément et avec une qualité sonore bornée par les capacités de l'oreille humaine. Bien que du son avec une qualité de CD audio soit tout à fait confortable pour des musiciens, nous estimons à 20ms le seuil de perception des décalages par l'oreille humaine. Dans ce cas,

l'instantanéité n'est pas possible sur un système (logiciel et réseau) dont la latence est supérieure à 20ms. C'est pour cette raison que nous assurons la simultanéité globale entre musiciens grâce à un mécanisme de synchronisation décrit dans [Bou., 2003; Bou. et al., 2004] et implémenté par *nJam* (network jam), un pluggin de *jMax*. Cette synchronisation assure donc aux musiciens que le retour du mixage global de la musique est identique pour tous. *nJam* intègre la diffusion du son en multicast avec RTP, la synchronisation des flux audio, mais aussi le paramétrage du décalage par les musiciens. Pour cela, ils spécifient un tempo, ainsi que le décalage en unités musicales comme un temps, une croche, une double croche, etc. Ce paramétrage leur permet d'avoir un retour décalé, mais à la fois synchronisé et en rythme par rapport à la musique qu'ils sont en train de jouer sur leur instrument.

En résumé, en terme de contraintes pour le système d'exploitation et pour le réseau, pour *n* sites hébergeant des musiciens, chaque instance de *nJam* va envoyer sur le réseau un seul flux audio et en recevoir *n*. *FTS* va donc devoir gérer au moins un composant correspondant à une entrée (micro ou instrument) et un composant pour la sortie de la carte son. Au niveau du protocole RTP, l'isochronisme des données audio est assuré grâce à un estampillage correspondant au numéro des échantillons audio. Chaque site est donc cadencé par sa propre horloge, celle de la carte son locale. A chaque top d'horloge sur chaque site, un échantillon de 16 bits est produit et un échantillon de chaque source est consommé. Les contraintes de délivrance temporelle de bout en bout sont donc cruciales, pour la partie réseau dans *nJam* ainsi que dans la partie système *FTS* et ALSA.

Pour une meilleure prise en compte de ces contraintes de temps, dans cet article nous allons ordonnancer les différents composants en utilisant une technique d'ordonnancement temps réel.

3. BOSSA

Le concert virtuel réparti est une application écrite en C dont l'environnement d'exécution est le système Linux. Au niveau système, cette application est confrontée au partage des ressources, notamment en termes d'accès au processeur et aux périphériques (adaptateur réseau et carte son). Cependant, un accès garanti à ces ressources est nécessaire périodiquement. Dans ce contexte, l'utilisation d'un système Linux temps réel (RT-Linux[4] ou RTAI) nous permettrait d'expérimenter des politiques

[4] http://www.fsmlabs.com

d'ordonnancement non disponibles sur le système Linux classique. Néanmoins, pour bénéficier de ces politiques, l'application cible doit respecter la structure des tâches temps réel et appeler les fonctions système particulières de la librairie de Linux temps réel. Pour éviter de modifier le code source relevant de la logique de l'application, nous avons choisi d'utiliser plutôt BOSSA. En effet, à notre connaissance, c'est la seule plate-forme qui peut intégrer des ordonnanceurs temps réel paramétrables au noyau Linux, permettant ainsi à des processus de Linux d'être ordonnancés selon une politique d'ordonnancement définie dans BOSSA. La seule intervention nécessaire dans le code source étant l'insertion d'un appel de fonction servant à attacher les processus à l'ordonnanceur choisi.

Avant de présenter la plate-forme BOSSA[5], nous présentons le langage dédié de BOSSA qui permet d'écrire des politiques d'ordonnancement.

3.1 Le langage dédié de BOSSA

La technique utilisée par BOSSA pour intégrer de nouvelles politiques d'ordonnancement dans un système d'exploitation existant est l'utilisation d'un langage dédié (DSL: Domain Specific Language). DSL est un langage de programmation qui fournit un haut niveau d'abstraction adapté au domaine étudié et qui permet la vérification et la génération de code intégré automatiquement dans le système cible.

Dans BOSSA, chaque politique d'ordonnancement est implantée comme un ensemble de gestionnaires d'événements écrits en DSL [Law. et al., 2004a, Law. et al., 2004b] et traduits en C grâce à un compilateur dédié.

Une politique d'ordonnancement de BOSSA déclare :

- un ensemble de structures de données
- un ensemble de gestionnaires d'événements et
- un ensemble de fonctions interface

permettant à l'utilisateur d'interagir avec l'ordonnanceur. Le tableau 1 montre quelques unes des déclarations de BOSSA pour implémenter la politique Linux 2.2. La déclaration `process` liste les attributs spécifiques à la politique d'ordonnancement et associés à chaque processus. Le champ `policy` définit la politique d'ordonnancement de Linux 2.2 c'est-à-dire une gestion FIFO ou Round Robin des processus temps réel et des processus non temps réel. Les autres champs de la structure `process` sont utilisés pour déterminer la priorité courante des

[5] http://www.emn.fr/x-info/bossa

processus associés. Finalement, la déclaration `ordering_criteria` spécifie le processus de calcul de la priorité relative d'un processus. Le tableau 1 contient aussi des exemples de gestionnaires d'événements de la politique Linux2.2. Par exemple, le gestionnaire d'événement `block.*` insert un processus cible dans la file des processus bloqués alors que le gestionnaire d'événement `unblock.*` déplace un processus de la file des processus bloqués vers la file des processus prêts.

Tableau 1 : *Déclarations et Gestionnaires d'événements de la politique Linux 2.2*

Déclarations	Gestionnaires d'événements

`type policy_t = enum {SCHED_FIFO, SCHED_RR, SCHED_OTHER}`
 	
`process = { policy_t policy; int rt_priority; time priority; time ticks; system struct ctx mm; }`
 	
`ordering_criteria = { highest rt_priority, highest ticks, highest ((mm==old_running.mm)?1:0) }`
	`On block.* { e.target => bocked; }`
	
	`On unblock.* { if (e.target in blocked) { e.target => ready; } }`

3.2 Du noyau Linux à BOSSA

BOSSA est une plate-forme ajoutée au noyau Linux pour le développement d'ordonnanceurs de processus [Bar. et al., 2002]. L'objectif de BOSSA est de simplifier la conception d'ordonnanceurs de sorte qu'un programmeur d'application puisse développer des politiques d'ordonnancement spécifiques sans être expert des systèmes d'exploitation.

Une politique d'ordonnancement BOSSA est implantée comme un module qui reçoit les changements d'état des processus sous la forme d'événements pour utiliser cette information et prendre des décisions d'ordonnancement. Préparer un noyau pour qu'il soit utilisable avec BOSSA nécessite l'insertion de ces événements à des points particuliers du noyau, appelés *points d'ordonnancement*. L'évolution du noyau Linux pour supporter BOSSA est assez complexe, pour diverses raisons.

D'abord, BOSSA doit être utilisable avec n'importe quelle version de Linux. Une solution basée sur les patches serait insuffisante car les numéros de ligne correspondant aux points d'ordonnancement ainsi que le code entourant ces points diffèrent d'une version de Linux à une autre. De plus, certains changements nécessaires pour supporter BOSSA dépendent des propriétés de flux de contrôle. Sans compter le fait que toute modification effectuée à la main sur plusieurs fichiers source (le code source de Linux est de l'ordre de 100Mo) peut générer des erreurs.

C'est alors que le principe de réécriture a été utilisé pour implémenter une fonctionnalité transversale qui contient une collection de fragments de code ainsi qu'une description formelle des points d'ordonnancement. Cette fonctionnalité correspond à un ensemble de règles de réécriture qui utilisent la logique temporelle pour décrire les conditions à vérifier en vue d'insérer des événements spécifiques. L'utilisation de la logique temporelle permet entre autres de résoudre le problème de sensibilité au contexte.

Voici un exemple de règle de réécriture tel qu'il est décrit dans [Aber. et al., 2003] :

```
n:(call try_to_wake_up))
     => Rewrite(n,bossa_unblock_process(args))
```

Cette règle sera appliquée à chaque fois qu'un appel à la fonction `try_to_wake_up` est trouvé. Cet appel sera référencé par *n*. L'utilisation de `Rewrite` indique que l'appel `try_to_wake_up` est remplacé par un appel à `bossa_unblock_process`. La fonction `wake_up_process` suivante illustre l'effet de l'application de la règle décrite ci-dessus:

```
wake_up_process(struct task_struct *p) {
#ifdef CONFIG_BOSSA
      return
        bossa_unblock_process(WAKE_UP_PROCESS,   p,
0);
#else
     return   try_to_wake_up(p,0);
#endif
}
```

Le noyau est réécrit à l'aide d'une quarantaine de règles logiques d'une assez grande complexité implémentées en Ocaml et en PERL par l'intermédiaire de CIL (C Intermediate Language). Même avec ces

méthodes qui sont censées garantir un minimum de fiabilité, l'erreur est toujours possible. Ainsi, dans le cadre de l'utilisation de BOSSA pour faire tourner le concert réparti, nous avons pu constater le manquement d'une règle de réécriture.

3.3. BOSSA : une hiérarchie d'ordonnanceurs

Un ordonnanceur est une application complexe puisqu'il est question de comprendre les conventions utilisées dans l'implémentation d'un système d'exploitation. Il est souvent nécessaire de bien connaître le noyau pour rédiger une nouvelle politique car celle-ci est souvent complètement indissociable du reste du noyau. Idéalement, pour pouvoir implanter de nouvelles politiques d'ordonnancement, l'ordonnanceur et le reste du noyau doivent être complètement dissociables mais parfaitement interfacés.

BOSSA propose un niveau d'abstraction spécifique au domaine de l'ordonnancement. Au lieu de faire directement appel à des fonctions de l'ordonnanceur (typiquement `schedule`()), les drivers font appel à un système d'événements. En effet, la réécriture sur le noyau Linux consiste à analyser le code par l'intermédiaire de règles logiques afin de détecter les points d'ordonnancement (par exemple élection d'un nouveau processus, changement d'état d'un processus, etc.). Chaque point d'ordonnancement est remplacé par le lancement d'un événement qui dépend des conditions d'appel. Les événements sont interprétés par le RTS (Run - Time System) de BOSSA (voir Figure 3) qui invoque la fonction appropriée définie par l'ordonnanceur.

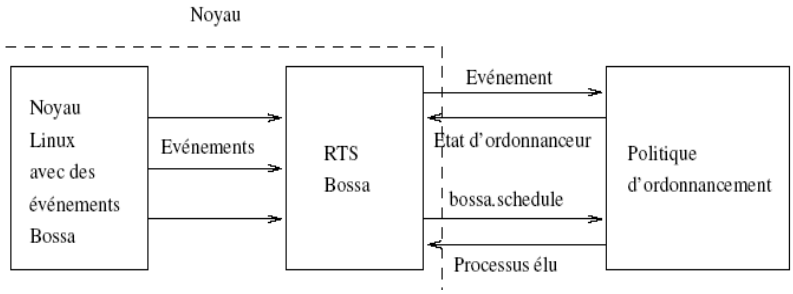


Figure 3. *L'architecture de BOSSA*

Pour régler les problèmes de cohabitation de programmes temps réel et non temps réel, BOSSA introduit la notion de hiérarchie d'ordonnanceurs. Un *ordonnanceur de processus* est un ordonnanceur classique qui gère les processus en vue de leur attribuer le processeur. Un *ordonnanceur virtuel* est un ordonnanceur qui contrôle d'autres

ordonnanceurs. Ainsi, on peut créer un ordonnanceur virtuel avec des ordonnanceurs fils auxquels il peut donner la main selon des critères bien définis (par exemple, priorité) ou bien selon une proportion (par exemple on donnera la main une fois sur trois à l'ordonnanceur fils numéro 1 contre deux fois sur trois à l'ordonnanceur fils numéro 2). L'ordonnanceur du système aura donc une forme hiérarchique ou arborescente avec comme feuilles des ordonnanceurs de processus et comme nœuds des ordonnanceurs virtuels.

Lorsque le RTS de BOSSA envoie un événement de type `bossa.schedule`, celui-ci est destiné au premier ordonnanceur dans la hiérarchie. Du point de vue de la programmation, un ordonnanceur virtuel va avoir pour tâche principale de propager l'événement reçu vers l'ordonnanceur fils approprié et de mettre à jour l'état de l'ordonnanceur fils. Ainsi, les événements descendent la hiérarchie jusqu'à atteindre un ordonnanceur de processus qui va, en traitant l'événement, agir directement sur les processus qu'il gère.

4. Ordonnancement temps réel du concert réparti

Les algorithmes d'ordonnancement temps réel décrits dans la littérature sont nombreux [Cot. et al., 2002] néanmoins ils ne sont pas implantés sur des systèmes d'exploitations usuels (non temps réel). Si notre choix s'est porté sur BOSSA c'est bien parce que notre application multimédia continuera à tourner sous Linux tout en utilisant une stratégie temps réel pour l'ordonnancement des processus.

Dans la suite de ce paragraphe, nous allons recenser les processus temps réel du concert réparti qu'il faudra ordonnancer en utilisant une politique d'ordonnancement temps réel. Pour cela, nous nous intéressons aux modules *FTS/jMax* et *nJam* qui constituent le cœur de notre application.

4.1 Analyse des processus de FTS et de nJam

Comme expliqué dans la section 2, au niveau de chaque site, *FTS* (le moteur de calcul audio) va gérer les composants *jMax* comme les entrées/sorties audio ou *nJam*. Lors de l'initialisation de *FTS,* les modules dépendants du système d'exploitation utilisé vont être chargés, par exemple le module ALSA sous Linux. Une fois ces modules chargés, l'utilisateur peut placer, relier et paramétrer les composants dans l'interface graphique.

Le moteur *FTS* est structuré en boucle. C'est au cours de cette boucle (voir ci-dessous) que vont être exécutées une par une les fonctions associées aux composants *jMax*. En effet, la fonction

`fts_sched_do_select` retourne dans `main_sched` la liste des fonctions à exécuter.

```
void fts_sched_run(void)
{
  while(main_sched.status != sched_halted)
    fts_sched_do_select(&main_sched);
}
```

FTS commence par analyser la sortie de l'interface graphique pour en déduire un treillis de dépendances entre les composants *FTS*. L'exécution des fonctions associées aux composants va permettre l'échange de données audio entre composants et éventuellement leur sortie sur la carte son. Cette boucle est critique puisque chaque fonction inscrite dans le moteur *FTS* correspond à un ensemble d'échantillons qui doivent être disponibles au prochain cycle. Lors de nos expériences, les cycles étaient de 64 échantillons, correspondant à une durée de 64/44100 secondes, soit 1.45 ms.

En plus de *FTS*, le patch *nJam* permet d'une part de synchroniser des musiciens entre eux et d'autre part de gérer les communications réseau. Le principe de *nJam* est de garantir d'une part l'isochronisme des échantillons audio provenant des différentes sources, et d'autre part de garantir la cohérence perceptive [Bou. et al., 2004] parmi les musiciens. La gestion de l'isochronisme des flux se fait grâce à une zone tampon qui absorbe la gigue du réseau. Ainsi, les paquets perdus ou tardifs sont remplacés par un paquet initialisé avec des zéros. Une phase de consensus entre les modules *nJam* des différents musiciens sur le vecteur des index de lecture des tampons permet d'assurer la cohérence perceptive. Cette phase préalable au jeu réparti est effectuée par échange de messages dans le champ APP (pour application) du protocole RTP [Sch. et al., 1998].

L'activation de *nJam* lance une thread qui boucle sur la réception et l'émission des paquets RTP. C'est l'opération la plus gourmande du point de vue ressources.

```
void start_routine(nJam_t *this)
{
    struct timeval  timeout;

    pid_t my_pid;
    my_pid = getpid();
```

```
    while(
         this->state == 1
         || this->state == NJAM_STATE_STOP)
{
          timeout.tv_sec  = 1;
          timeout.tv_usec = 0;
          if(                         rtp_recv(this-
>session,&timeout,this->rtp_ts_send))
           {
            rtp_send_ctrl(this->session,      this-
>rtp_ts_send,
               self_sync_packet_craft);
           }
          rtp_update(this->session);
        }

    this->thread_closed=1;
    pthread_exit(0);
}
```

4.2 Le concert réparti sous BOSSA

Afin d'intégrer BOSSA dans l'exécution du concert réparti, nous définissons une hiérarchie d'ordonnanceurs. L'équipe de BOSSA[6] a déjà travaillé sur une application multimédia [Conc. et al., 1998], *mplayer* pour « The Movie Player ». Ils ont choisi d'attacher cette application à un ordonnanceur EDF. Cette politique d'ordonnancement nécessite la définition de paramètres temporels telles que la période et la durée d'exécution exprimées en jiffies (tops d'horloge CPU). Nous définissons une arborescence à un seul niveau de sorte que la racine qui correspond à un ordonnanceur virtuel soit composé de deux ordonnanceurs de processus : un qui correspond à la version EDF de *mplayer* et l'autre à un ordonnanceur de processus classique de Linux. L'ordonnanceur virtuel est à priorité fixe, autrement dit manipule deux ordonnanceurs ayant des priorités statiques. Dans notre cas, les priorités sont associées aux ordonnanceurs de processus de manière à privilégier systématiquement EDF à un ordonnanceur Linux. Les commandes suivantes ont permis la création sous BOSSA de la hiérarchie d'ordonnanceurs de la Figure 4.

```
panoramix:/home/cordry # modprobe EDFu
panoramix:/home/cordry # modprobe Fixed_priority
panoramix:/home/cordry # ./bin/manager
Available schedulers:
```

[6] http://www.emn.fr/x-info/bossa/

```
0. Linux (PS, root, default)
1. EDFu (PS, not loaded, not default)
2. Fixed_priority (VS, not loaded, default)

Default path:
Linux

Command: (use the scheduler number)
  c <P> <C>     connect parent scheduler P to
child scheduler C
  d <S>         disconnect scheduler S
  l             list available schedulers
  h             print this help menu
  q             quit

> c 2 0
int importance_10: 5
> c 2 1
int importance_10: 7
> l
Available schedulers:
0. Linux (PS, loaded, default)
1. EDFu (PS, loaded, not default)
2. Fixed_priority (VS, root, default)

Default path:
Fixed_priority -> Linux
> q
```

Tous les processus vont par défaut s'exécuter sous Linux excepté pour la boucle principale de *FTS* (qui fait du calcul audio) et la thread *nJam* (chargée des émissions et des réceptions des paquets RTP) qui vont être ordonnancées en temps réel.

Tout processus qui doit être ordonnancé sous BOSSA, doit être attaché à une politique d'ordonnancement. Dans notre cas, la boucle *FTS* va être attachée à l'ordonnanceur EDF en spécifiant la durée maximale d'exécution (wcet[7]) de la boucle *FTS* ainsi que son échéance égale à la période.

[7] worst case execution time

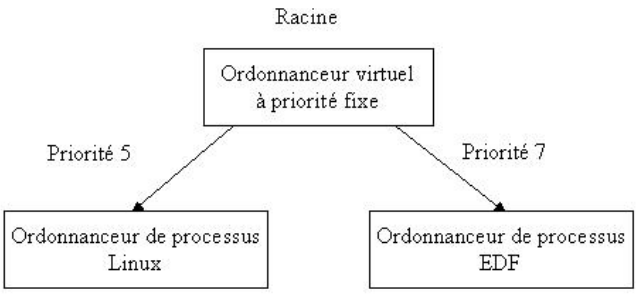


Figure 4. *Hiérarchie utilisée*

Pour calculer la durée d'exécution de la boucle *FTS*, il faut évaluer le temps d'exécution des différentes fonctions appelées. Dans nos expériences, ces fonctions associées aux composants permettent de préparer 64 échantillons à chaque top d'horloge de la carte son, soit un cycle qui prend 64/44100=1.45 ms. Plus le nombre de composants est important, plus le nombre de fonctions associées sera important, ce qui augmente la durée d'exécution de la boucle *FTS*. Dans nos expériences, selon le nombre de composants définis nous avons borné la durée d'exécution de la boucle FTS à 4 jiffies (tops d'horloge CPU, sur un matériel moderne un jiffie fait environ 10 ms) et fixé sa période à 5 jiffies.

Le code suivant montre les modifications qui ont été effectuées sur la boucle FTS afin de l'attacher à l'ordonnanceur EDF.

```
/* On inclut les definitions liees a EDFu */
#include "user_stub_EDFu.h"

/* [...] */

void FTS_sched_run(void)
{
  int period = 5;
  int wcet = 4;

  /*   On   attache   le   processus   courant   à
l'ordonnanceur EDFu */
  if (EDFu_attach(0,period,wcet) < 0)
    FTS_post(  "Cannot  attach  (%s)\n",  strerror(
errno));

  while(main_sched.status != sched_halted)
    FTS_sched_do_select(&main_sched);
```

```
    /* On parcourt en boucle la liste de fonctions
appelées
       par FTS jusqu'à l'arrêt de celui-ci
    */
}
```

De la même manière, nous avons attaché la thread *nJam* qui est beaucoup moins gourmande en ressources puisqu'elle ne gère pas directement le son mais s'occupe uniquement de recevoir et d'émettre des paquets RTP. On se contente de lui associer une période de 10 jiffies, pour une durée maximale d'exécution de 1 jiffie. Le code suivant montre l'attachement de la thread *nJam* à l'ordonnanceur EDF et la Figure 5 montre l'ordonnancement des processus *FTS* et *nJam* en utilisant la politique EDF.

Attachement de nJam au scheduler

```
void start_routine(nJam_t *this)
{
    struct timeval  timeout;

  pid_t my_pid;
  int wcet = 1;
  int period = 10;
  my_pid = getpid();
  if (EDFu_attach(0,period,wcet) < 0)
    fts_post("Cannot           attach           %u
(%s)\n",my_pid,strerror( errno));
    fts_post("start routine \n");
    ...
    pthread_exit(0);
}
```

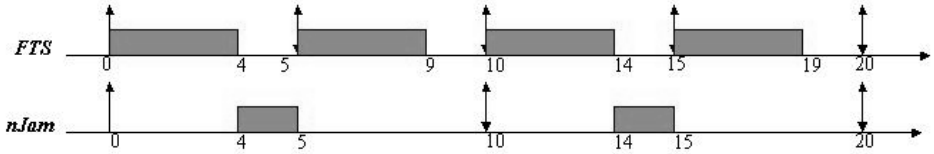


Figure 5: *Diagramme de Gantt avec un ordonnancement EDF*

5. Calcul des Performances

Nous avons fait fonctionner le concert réparti en utilisant des automates son qui génèrent un signal audio PCM 16 bit à une fréquence

de 44100 Hz (un sur chaque site). La machine appelée "*breton*" a un processeur Intel de 3 GHz avec 1 Go de mémoire et fonctionne avec un système Linux (noyau 2.6.5). La machine "*panoramix*" a un processeur Intel de 350 MHz avec 256 Mo de mémoire et fonctionne avec un système Linux dont deux noyaux ont été installés: celui de BOSSA et celui de linux (version 2.4.21). Les courbes que nous allons présenter correspondent à la quantité de données présentes dans les tampons de *nJam* en fonction du temps. Ces tampons sont alimentés par le réseau à raison d'un tampon pour chaque source musicale. Ces données sont régulièrement consommées par *FTS* afin d'alimenter la carte son. Étant donné que la production d'échantillons audio ainsi que leur consommation ont lieu théoriquement à la même cadence (si on considère que les horloges des cartes son ne dérivent pas), la quantité de données devrait être constante, modulo la gigue du réseau. Les courbes sont prises depuis l'origine de la communication entre les machines, ce qui explique la brusque augmentation qui est causée par l'ajustement des latences de restitution pour la synchronisation des flux audio.

La Figure 6 représente la situation idéale pour la machine *panoramix* (qui fonctionne avec le noyau BOSSA), c'est-à-dire lorsque le système n'est pas chargé. Ici, la relation producteur/consommateur des deux flux audio provenant de *panoramix* et de *breton* est correctement conservée puisque les courbes restent stables à partir de la 33ème seconde, moment auquel l'ajustement des latences a été correctement établie. La stabilité de cette courbe montre que nous arrivons à conserver une latence fixe dans notre mécanisme de streaming. Ici, la latence correspond à une mesure à quatre temps dans un tempo de 120 BPM (battements par minutes), soit 2 secondes (valeur configurée par l'utilisateur).

La figure 7 nous montre le cas où la machine *panoramix* fonctionne avec le noyau linux chargé à l'aide du script décrit en figure 7 et lancé à la 29ème seconde sur *panoramix*. Dès cet instant, les échantillons audio ne sont plus entendus en sortie de la carte son, nous laissant supposer que celle-ci n'est plus alimentée, causant un déséquilibre dans la relation producteur/consommateur dans *nJam*. La courbe de la figure 7 nous montre la taille des tampons de *nJam* dans la machine *panoramix*. Nous pouvons voir que les données venant de *breton* ne sont pas consommées puisque leur quantité ne cesse de croître. Cependant, le flux local reste constant, nous laissant alors supposer que les données ne sont pas envoyées. Cette supposition est confirmée par la courbe de la figure 8, car la machine *breton* cesse brusquement de recevoir des données de *panoramix* à la 50ème seconde. Nous voyons donc clairement grâce à la figure 7 que la charge CPU paralyse complètement l'accès à la carte son

(processus *FTS*) ainsi que l'envoi et la réception des données sur le réseau (thread *nJam*).

Nous avons fait le même test de charge avec *panoramix* fonctionnant sous BOSSA. Le processus *FTS* ainsi que la thread *nJam* étant ordonnancés avec une politique EDF. Dans ce cas, malgré la charge, nous pouvons voir sur la figure 9 que la machine *panoramix* n'est pas affectée par le script de charge, ni en terme de réseau, ni en terme d'accès à la carte son.

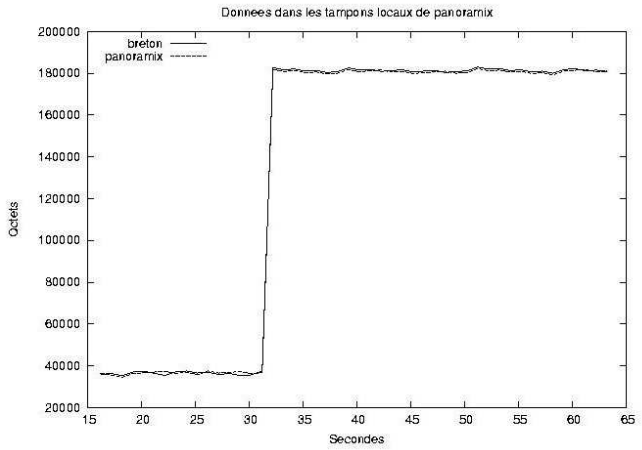
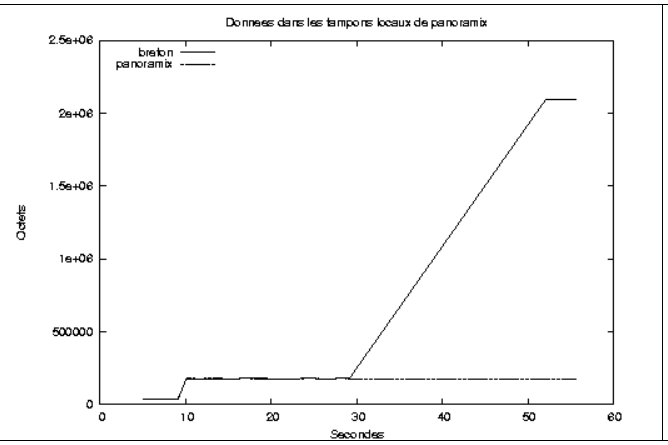


Figure 6. Comportement de *panoramix* sous BOSSA et de *breton* sous Linux lorsque le système n'est pas chargé (du point de vue de *panoramix*)



Script :

```
#!/bin/bash
while true;
do r=5;
done
```

Figure 7. Tailles des tampons de *panoramix* et de *breton* lorsqu'elles sont sous Linux avec un système chargé (du point de vue de *panoramix*)

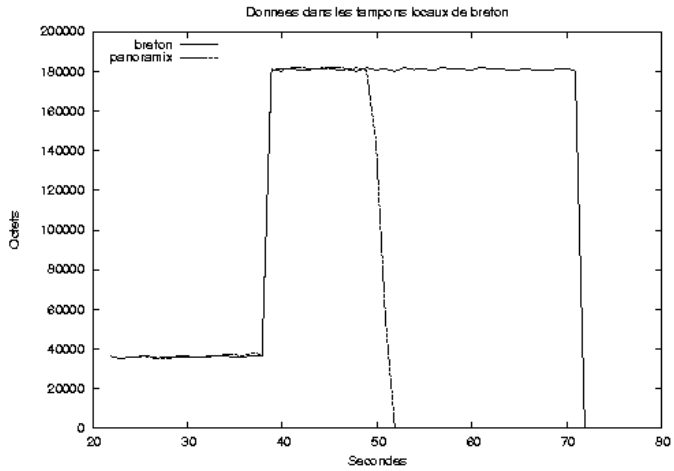


Figure 8. Tailles des tampons de *panoramix* et de *breton* lorsqu'elles sont sous Linux avec un système chargé (du point de vue de *breton*)

6. Conclusion

Le projet portant sur le concert virtuel réparti a pour objectif de fournir des moyens à des musiciens physiquement éloignés pour jouer via Internet. Dans cet article, on s'est attaché à résoudre l'ordonnancement local des processus de notre application multimédia.

Notre prototype est composé de deux parties, le moteur de son audio (*FTS/jMax*) et une partie réseau (*nJam*, un pluggin de *FTS*) qui gère les communications et la cohérence perceptive [Bou. et al., 2004] des flux audio parmi les musiciens.

Nous avons voulu montrer ici comment nous avons procédé pour ordonnancer les différents processus générés par ces composants en utilisant une technique d'ordonnancement temps réel. Il en résulte une meilleure prise en compte des contraintes de temps de l'application. Nous avons utilisé BOSSA, une plate-forme orientée événements qui permet d'intégrer de manière très souple de nouvelles politiques d'ordonnancement organisées dans une hiérarchie arborescente d'ordonnanceurs. Nous avons alors pu faire cohabiter l'ordonnancement Linux et un ordonnanceur EDF, auquel nous avons attaché notre application multimédia. Les processus non temps réel étant attachés à l'ordonnanceur de Linux.

Sans changer de système d'exploitation, mais en intégrant BOSSA, nos expérimentations ont montré que notre application respectait ses contraintes temporelles, notamment en terme de latence constante pour les moteurs réseau et son.

Plusieurs pistes de recherche peuvent constituer une perspective à ce travail. Particulièrement:

- D'une part l'utilisation d'appels systèmes dans le paramétrage de l'ordonnanceur. En effet, localement, nos contraintes temporelles s'expriment en cycles d'accès à la carte son, et pas en accès processeur (jiffies).
- d'autre part l'étude précise des paramètres relatifs à la qualité de service du réseau qui pourraient influer sur les caractéristiques temporelles des processus temps réel du concert réparti.

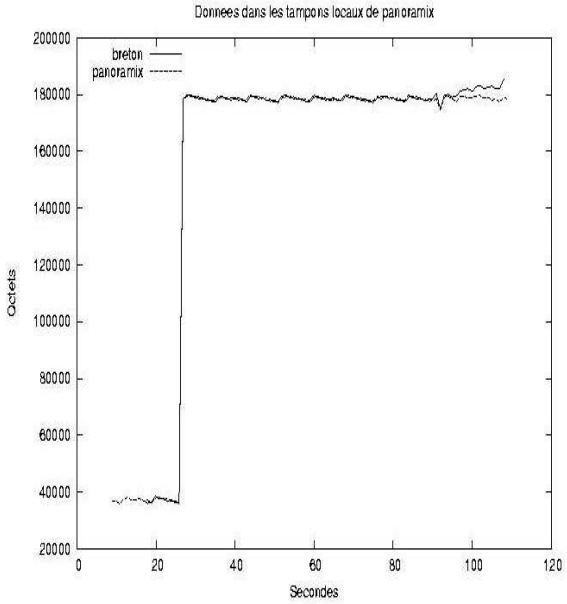


Figure 9. Tailles des tampons de *panoramix* fonctionnant sous BOSSA avec un système chargé

Bibliographie

[Aber. et al., 2003] : Aberg R. A., Lawall J.L., Südholt M., Muller G. And Le Meur A.-F., "*On the automatic evolution of an OS kernel using temporal logic and AOP*", Automated Software Engineering, 2003.

[Bar. et al., 2002] : Baretto L. P. et Muller G., "*Bossa: a language-based approach to the design of real-time schedulers*", In 10th International Conference on Real-Time Systems (RTS'2002), pages 19-31, Paris, France,march 2002.

[Bou., 2003] : Bouillot N., "*Un algorithme d'auto synchronisation distribuée de flux audio dans le concert virtuel réparti*", RenPar'15, CFSE'2003, SympAAA'2003, pp.441-452, France, oct. 2003.

[Bou. et al., 2004] Nicolas Bouillot N. et Gressier-Soudan E.,"*Consistency models for Distributed Interactive Multimedia Applications*". A paraître dans Operating Systems Review. Volume 38, issue 3. Octobre 2004.

[Conc. et al., 1998] : Consel C. et Marlet R., "*Architecturing software using a methodology for language development*", Proc. of the 10th Intern. Symp. On Programming Languages, Implementations, Logics and Programs, Pise, Italy, pp. 170-194, 1998.

[Cot. et al., 2002] : Cottet F., Delacroix J., Kaiser C. et Mammeri Z., "Scheduling in Real-Time Systems", Wiley Ed., 261 pages, 2002.

[Déch., 2000] : Déchelle F., "*jmax: un environnement pour la réalisation d'applications musicales temps réel sous Linux*", Actes des journées d'Informatique Musicale, 2000.

[Law. et al., 2004a] : Julia L. Lawall, Gilles Muller, Hervé Duchesne, "*language design for implementing process scheduling hierarchies*", Invited application paper, in Proc. of the ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 80-91, ISBN:1-58113-835-0, Italy, August 24-25, 2004.

[Law. et al, 2004b] : Julia Lawall, Gilles Muller, Anne-Francoise Le Meur, " *On the design of a domain-specific language for OS process-scheduling extensions*", in Proc. of the Third International Conference on Generative Programming and Component Engineering (GPCE'04), Vancouver, October 24-28, 2004.

[Loc. et al., 2003] Locher H.-N., Bouillot N. Becquet E., Déchelle F. & Gressier-Soudan E.,"*Monitoring the Distributed Virtual Orchestra with a CORBA based Object Oriented Real-Time Data Distribution Service*", International Symposium on Distributed Object Application, nov. 2003. Catagne, Italy.

[Sch. et al., 1998] Schulzrinne, Casner, Frederick and Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889. 1998.

Remerciements :

Nous tenons à remercier M. Gilles Muller, Professeur à l'Ecole des Mines de Nantes et membre de l'équipe de BOSSA, ainsi que M. Jean Ferdinand Susini Maître de Conférences au CNAM, pour leurs remarques pertinentes qui ont permis d'améliorer le contenu de cet article.