MidiShare : une architecture logicielle pour la musique

Dominique Fober, Stéphane Letz, Yann Orlarey

Chapitre 5

MidiShare : une architecture logicielle pour la musique

Le développeur d'une application musicale est souvent confronté à des problèmes difficiles à résoudre, notamment parce qu'ils sont relatifs à la maîtrise du temps. Le manque de support des systèmes d'exploitations courants, l'absence de standard, les problèmes de portabilité qui en résultent ne facilitent pas la tâche du programmeur. Nous présentons MidiShare, une architecture logicielle qui a été conçue dans le but de couvrir les besoins des applications musicales de manière homogène, durable et portable. Nous montrons également à travers plusieurs exemples, comment cette architecture facilite le développement, notamment grâce à des mécanismes simples et efficaces de gestion du temps et de communication en temps réel.

5.1. Introduction

Le système d'exploitation d'un ordinateur fournit les services logiciels indispensables à son utilisation. Il organise le partage et la gestion des ressources nécessaires au fonctionnement des applications. Il facilite le travail du programmeur en lui proposant une « machine virtuelle » plus homogène et plus simple à programmer que la « machine réelle » sous-jacente. Il assure une plus grande pérennité aux applications en les isolant autant que possible des particularismes matériels. Ces rôles essentiels du système d'exploitation ne sont malheureusement que très partiellement remplis vis-à-vis des applications musicales.

Chapitre rédigé par Dominique FOBER, Stéphane LETZ, Yann ORLAREY.

Les applications musicales doivent ordonnancer et restituer les événements musicaux avec une grande précision temporelle, de l'ordre de la milliseconde. Elles doivent également travailler en collaboration avec d'autres outils multimédias et satisfaire à différents codes de synchronisation. De ce point de vue, la conception même des systèmes d'exploitation préemptifs multitâches ne permet pas de garantir l'éligibilité d'un processus à une date donnée, ce qui complique singulièrement le développement de processus qui requièrent une grande précision temporelle.

Du fait de contraintes temps réel et d'efficacité, les besoins des applications musicales en termes de mémoire sont également spécifiques et les gestionnaires fournis par les systèmes d'exploitation inadaptés :

– le temps d'allocation d'un bloc mémoire n'est pas déterministe, il peut être long si la mémoire a besoin d'être compactée ;

– pour chaque bloc alloué, il y a un surcoût de plusieurs octets qui rend l'allocation de petits blocs peu rentable, notamment pour les messages MIDI (3 octets de manière générale).

Enfin, les solutions proposées en termes de communication inter applications par les systèmes d'exploitation ne prennent pas en compte la dimension du temps propre aux applications musicales.

La nécessité de recourir à des architectures matérielles et logicielles spécialisées est donc apparue très tôt dans le domaine de l'informatique musicale. Dès la fin des années 1980, apparaissent des architectures ayant pour ambition de répondre à cette nécessité. Ce sont par exemple, *MROS*[1] ou *MIDI Manager* [MM 90], suivis un peu plus tard de *OMS* [OMS 95] ou FreeMIDI[2]. Nous présentons *MidiShare* [ORL 89], un système conçu par Grame en 1989, qui a entre autres la particularité d'être multiplates-formes et qui a évolué jusqu'à aujourd'hui alors que les systèmes cités ci-dessus ont quasiment tous disparus. Nous montrerons ensuite, à travers des exemples simples, comment les problèmes couramment rencontrés dans le cadre du développement d'applications musicales peuvent être aisément résolus grâce au support de cette architecture. Pour finir, nous aborderons quelques points délicats dans la conception d'applications temps réel.

5.2. MidiShare : une architecture logicielle pour la musique.

MidiShare est un système d'exploitation musical, MIDI, multitâches, temps réel et multiplates-formes, qui vient se mettre en complément du système d'exploitation

1. MROS (*MIDI Realtime OS*), développé pour Atari par Steinberg GmbH.
2. *FreeMIDI*, développé par Mark of the Unicorn Inc.

de la machine hôte. Il supporte les principales plates-formes logicielles : GNU/Linux, MacOS et Windows.

5.2.1. *Le modèle conceptuel*

MidiShare est basé sur un modèle client/serveur. Il est composé de six éléments principaux (figure 5.1) : un gestionnaire de mémoire, un gestionnaire du temps associé à un synchroniseur, un gestionnaire des tâches, un gestionnaire de la communication, un gestionnaire de drivers et un ordonnanceur.

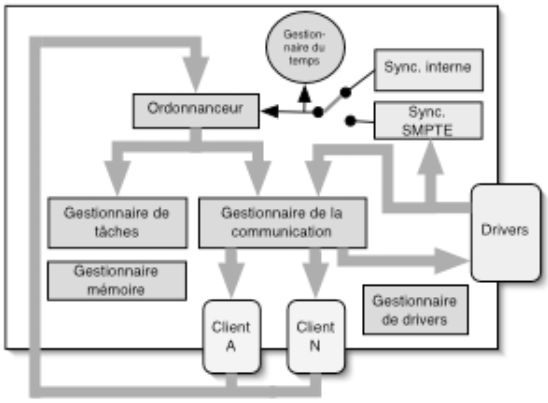

Figure 5.1. *Architecture générale de MidiShare*

5.2.1.1. *Le gestionnaire de mémoire*

La mémoire de MidiShare est basée sur des événements structurés, typés et datés avec une résolution à la milliseconde. Ils recouvrent à la fois les typologies MIDI et MIDI File. Le gestionnaire de mémoire fournit des primitives simples pour l'allocation, la copie et la libération de ces événements, avec un coût d'allocation borné et faible. Il a été conçu pour une utilisation temps réel, il met notamment en œuvre des techniques d'accès concurrent *lock-free* [FOB 02].

5.2.1.2. *L'ordonnanceur*

Il délivre les événements et les tâches à leurs dates d'échéance. Il est possible d'y insérer des événements ou des tâches pour une date quelconque dans le futur. L'algorithme d'ordonnancement [ORL 90] assure un coût de traitement borné et constant par événement, quelque soit la charge de l'ordonnanceur.

5.2.1.3. *Le gestionnaire de la communication*

La communication est basée sur des événements MidiShare, plus simples à manipuler que des paquets d'octets MIDI. Le gestionnaire de communication distribue ces événements, reçus de l'ordonnanceur et des drivers d'entrée, aux applications clientes et aux drivers de sortie, conformément aux connexions courantes.

5.2.1.4. *Le gestionnaire du temps et le synchroniseur*

Le gestionnaire de temps a pour charge de maintenir la date courante du système. Il a une résolution d'une milliseconde. Le synchroniseur supporte le code SMPTE de manière transparente pour les applications clientes.

5.2.1.5. *Le gestionnaire de tâches*

Il a pour responsabilité d'activer pour le compte des applications clientes, les tâches qui sont délivrées à leur date d'échéance par l'ordonnanceur.

5.2.1.6. *Le gestionnaire de drivers*

Les drivers ont un statut particulier puisqu'ils peuvent être intégrés dynamiquement au noyau. C'est le gestionnaire de drivers qui a pour responsabilité de les charger et de les activer.

5.2.2. *Des mécanismes sophistiqués pour la gestion du temps.*

Les applications musicales requièrent une gestion sophistiquée du temps : il ne leurs suffit pas de pouvoir réagir en temps réel aux événements entrants, elles doivent également planifier le futur avec une grande précision. MidiShare fournit plusieurs mécanismes simples et puissants pour la gestion du temps : ce sont les alarmes et les appels de fonction différés dans le temps.

5.2.2.1. *Les alarmes*

5.2.2.1.1. Alarmes de réception

Chaque application cliente peut installer une alarme de réception qui lui permettra de traiter les événements reçus en temps réel. Cette alarme sera activée par MidiShare pour le compte de l'application, au moment où le gestionnaire de communication lui aura distribué un ou plusieurs événements.

5.2.2.1.2. Alarmes de contexte

Chaque application cliente peut installer une alarme de contexte qui lui permettra de réagir aux modifications de son environnement en temps réel. MidiShare active les alarmes de contexte au moment où le changement de contexte s'effectue. Ils

peuvent concerner par exemple la modification des connexions entre applications ou encore l'ouverture d'un nouveau client.

5.2.2.2. *Les appels de fonction différés dans le temps*

5.2.2.2.1. Tâches temps réel

Une application peut effectuer des appels de fonction différés dans le temps en créant une tâche et en confiant son appel à MidiShare pour une date donnée. Lorsque cette tâche arrive à échéance, elle est rendue par l'ordonnanceur et activée par le gestionnaire de tâches pour le compte de l'application qui l'a créée. Cet appel de fonction se fait en temps réel.

5.2.2.2.2. Tâches temps différé

Un mécanisme analogue permet d'effectuer des appels de fonctions en temps différé. A la différence des tâches temps réel, une tâche temps différé, rendue par l'ordonnanceur, n'est pas exécutée par le gestionnaire de tâches mais placée dans la liste des tâches à exécuter de l'application. Son appel est alors de la responsabilité de l'application qui l'a créée. Ce mécanisme permet entre autres d'ordonnancer des tâches qui n'ont pas de contraintes temps réel fortes.

5.2.3. *Une boite à outils canonique*

MidiShare fournit au développeur, un ensemble simple et concis de fonctions qui permettent de satisfaire à l'ensemble des besoins de base des applications musicales. Ces fonctions sont disponibles dans une grande variété de langages (C, C++, Lisp, Java) et de manière uniforme sur les plates-formes logicielles supportées. Les services correspondants peuvent être regroupés dans les catégories suivantes.

5.2.3.1. *Gestion des sessions*

En premier lieu, une application doit s'assurer que MidiShare est installé en mémoire, ce qui est fait par l'appel de la fonction `MidiShare`. Si MidiShare est présent, l'application peut alors ouvrir une ou plusieurs sessions avec la fonction `MidiOpen` qui renvoie un identifiant unique qui servira tout au long de la durée de la session. Toute session ouverte devra être refermée avec la fonction `MidiClose` avant la fin de l'application.

5.2.3.2. *Communication et connexions*

Pour qu'un client MidiShare puisse transmettre et recevoir des événements, il doit tout d'abord être connecté à une ou plusieurs sources et/ou destinations. Un client peut être vu comme une *boite noire*, recevant un flot d'événements en entrée

et produisant un flot d'événements en sortie. Cette *boite noire* peut être librement connectée à d'autres, permettant ainsi de former des réseaux de communication arbitrairement complexes. La communication avec les drivers se fait par le biais d'un *pseudo-client* portant le nom « MidiShare » et ayant le numéro de référence 0. Pour communiquer avec le monde extérieur, un client doit donc être connecté à ce *pseudo-client*.

L'établissement ou la suppression de connexions se fait simplement avec la fonction `MidiConnect` qui prend comme argument les numéros de référence de la source et de la destination, ainsi que l'état désiré de la connexion. La fonction `MidiIsConnected` permet d'interroger l'état de la connexion entre deux clients. Il n'y a pas de restriction sur le nombre de connexions qu'un client peut établir avec d'autres.

Dans certains cas, notamment si l'on veut écrire une application de gestion des connexions, il est important de pouvoir obtenir des informations concernant l'ensemble des clients du système :

– la fonction `MidiCountAppls` permet de connaître le nombre de clients ;

– les fonctions `MidiGetIndAppl` et `MidiGetNamedAppl` renvoient le numéro de référence d'un client en fonction de son numéro d'ordre et de son nom ;

– la fonction `MidiSetName` permet de changer le nom d'un client.

Enfin, un client peut être informé en temps réel des modifications de contexte du système (telles qu'un changement de connexion, l'ouverture ou la fermeture d'un client) : il lui suffit de définir une fonction particulière de gestion des alarmes de contexte, qui pourra être installée avec `MidiSetApplAlarm`. MidiShare se chargera d'activer cette fonction lors de chaque changement de contexte.

5.2.3.3. *Emission et réception*

Une fois connecté, un client peut recevoir et émettre des événements MidiShare. Chaque client possède un FIFO de réception dans lequel MidiShare dépose une copie des événements qui lui sont destinés. Ces événements peuvent provenir d'autres clients ou encore des drivers. La fonction `MidiCountEvs` permet à tout moment de connaître le nombre d'événements en attente dans le FIFO de réception. Ces événements peuvent être retirés du FIFO par des appels successifs à la fonction `MidiGetEv`. Enfin, la fonction `MidiFlushEvs` détruit tous les événements en attente. Les événements retirés du FIFO appartiennent au client correspondant : il peut en disposer librement, mais il est de sa responsabilité de le libérer quand il n'en a plus besoin.

Chaque client peut sélectionner les événements qu'il désire recevoir en utilisant un filtre. Ce filtre agit localement pour son client et n'a donc aucune influence sur les événements reçus par d'autres clients. La gestion de ces filtres se fait avec les fonctions `MidiSetFilter` et `MidiGetFilter`.

Le gestionnaire de temps de MidiShare maintient une horloge interne d'une longueur de 32 bits. Cette horloge est utilisée tant pour dater (en millisecondes) les événements reçus, que pour spécifier la date d'émission d'un événement. Cette horloge peut être interrogée avec la fonction `MidiGetTime`.

Trois fonctions permettent d'émettre des événements :

– `MidiSendIm` : pour la transmission immédiate d'un événement,

– `MidiSend` : pour la transmission d'un événement à sa date courante,

– `MidiSendAt` : pour la transmission d'un événement à une date donnée.

MidiShare se charge automatiquement de délivrer les événements à leur date d'échéance. Grâce à ce mécanisme, un client peut planifier la transmission d'un événement plusieurs jours à l'avance avec une résolution à la milliseconde. Tout événement émis par un client (avec les fonctions `MidiSend`, `MidiSendAt` ou `MidiSendIm`) n'est plus accessible à ce client : il ne doit plus y faire référence sous peine de désorganiser gravement la mémoire MidiShare.

5.2.3.4. *Gestion de la mémoire*

La mémoire MidiShare est organisée en cellules de taille fixe (16 octets) qui permettent de constituer des événements structurés. La fonction `MidiNewEv` permet d'allouer un événement du type désiré. La fonction `MidiFreeEv` permet de le libérer. Un événement alloué peut être dupliqué avec la fonction `MidiCopyEv`. Enfin, l'espace mémoire libre peut être interrogé avec `MidiFreeSpace`, ou étendu avec `MidiGrowSpace`.

Chaque événement est composé d'une cellule d'en-tête qui peut être suivie d'une ou plusieurs cellules d'extension. La figure 5.2 décrit les champs de la cellule d'en-tête :

– le champ `link` est utilisé de manière interne pour chaîner des cellules ;

– le champ `date` contient la date de l'événement ;

– le champ `evType` indique le type de l'événement ;

– le champ `refNum` contient le numéro de référence du client émetteur ;

– le champ `port` indique le port le destination de l'événement ;

– le champ `chan` indique le canal MIDI de destination de l'événement.

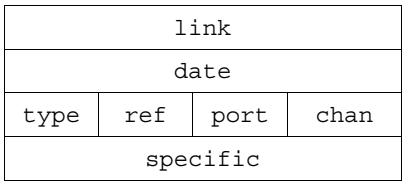

Figure 5.2. *Structure d'une cellule d'en-tête*

Quelque soit le type de l'événement, ces champs sont toujours présents et on peut y accéder directement. Le dernier champ `specific` de la cellule est par contre dépendant du type de l'événement. Dans certains cas, il peut contenir un pointeur sur une ou plusieurs cellules d'extension. Un accès direct à cette partie de la cellule suppose de prendre en compte la structure mémoire correspondant au type de l'événement. C'est pourquoi les fonctions `MidiGetField` et `MidiSetField` permettent de cacher cette structure interne en accédant aux champs spécifiques par indice entre 0 et `MidiCountFields` - 1.

Un certain nombre d'événements ont un nombre de champs variable, c'est le cas par exemple des *System Exclusive*, dans ce cas la fonction `MidiCountFields` renvoie le nombre de champs de la partie variable de l'événement et la fonction `MidiAddField` permet d'ajouter un champ à la fin de la partie variable.

5.2.3.5. *Les séquences MidiShare*

MidiShare permet également la gestion de séquences d'événements ordonnés dans le temps. La fonction `MidiNewSeq` alloue une nouvelle séquence d'événements initialement vide et la fonction `MidiAddSeq` insère un événement dans cette séquence en maintenant l'ordre temporel. La fonction `MidiClearSeq` permet de vider une séquence de son contenu en libérant les événements et la fonction `MidiFreeSeq` libère à la fois la séquence et les événements qu'elle contient.

5.2.3.6. *Les tâches temps réel*

MidiShare fournit à ses clients un moyen simple d'être informé en temps réel de l'occurrence d'un événement par le biais des *alarmes*. Une alarme est une fonction dont l'adresse est passée à MidiShare par un client. Cette fonction sera activée en temps réel par le système lors de l'occurrence d'un événement.

Deux types d'alarmes couvrent les besoins des clients : la première couvre les changements globaux de contexte du système (voir paragraphe 5.2.3.2) et peut être activée avec la fonction `MidiSetApplAlarm`. La seconde est activée avec

`MidiSetRcvAlarm` et informe un client de la présence de nouveaux événements dans son FIFO de réception. Une alarme est toujours activée sous interruption, un client doit donc prendre garde à ne pas faire appel aux ressources du système d'exploitation qui ne sont pas protégées contre la réentrance. A part `MidiOpen` et `MidiClose`, l'ensemble des fonctions de MidiShare sont disponibles dans le contexte des alarmes. Un client peut également accéder à ses variables globales puisque MidiShare restaure le contexte de l'application avant d'activer une alarme.

MidiShare implémente un second mécanisme pour gérer les tâches temps réel : il s'agit des appels de fonctions différés dans le temps. Les fonctions `MidiTask` et `MidiDTask` prennent une date, une fonction et ses paramètres comme arguments, MidiShare crée un événement particulier (de type `typeProcess` ou `typeDProcess`) à partir de ces arguments et passe cet événement à l'ordonnanceur.

A sa date d'échéance, MidiShare active la fonction pour le compte du client correspondant quand il s'agit d'un événement de type `typeProcess`. Pour les événements de type `typeDProcess`, la tâche est mise en attente dans une liste appartenant au client : la fonction `MidiCountDTasks` permet à un client de connaître le nombre de tâches en attente et le cas échéant, la fonction `MidiExec1Dtask` exécute la prochaine tâche en attente.

Dans certaines circonstances, il peut être utile *d'oublier* une tâche programmée mais non encore exécutée grâce à la fonction `MidiForgetTask`. Enfin, la liste des tâches en attente de type `typeDProcess` peut être vidée avec la fonction `MidiFlushDTasks`.

5.3. Exemples

5.3.1. *Un squelette typique*

Une application cliente de MidiShare va typiquement commencer par configurer une session, exécuter ensuite le corps du programme et refermer la session avant de quitter (exemple 5.1). Elle gère des variables globales, dont le numéro de référence du client MidiShare qui servira tout au long de la session.

La configuration d'une session passe par la vérification de la présence de MidiShare, l'ouverture de la session, le positionnement éventuel d'alarmes de réception et/ou d'application, l'installation optionnelle d'un filtre et enfin l'établissement de connexions (exemple 5.2).

```
#include <MidiShare.h>

short myRefNum ; /* numéro de référence du client
MidiShare */
MidiFilterPtr  myFilter=0 ; /* un filtre d'événements
(optionnel) */
MidiName       ClientName = "foo" ; /* le nom du client
*/

int main( int argc, char *argv[])
{
 /* configure une session MidiShare */
 if ( !SetUpMidi() ) return 0 ;
 /*
       corps du programme
 */
 /* referme la session */
 QuitMidi() ;
}
```

Exemple 5.1.

```
Boolean SetUpMidi(void)
{
 /* on vérifie que MidiShare est présent */
 if ( !MidiShare() ) return false ;

 /* on ouvre une session MidiShare */
 myRefNum = MidiOpen (ClientName) ;
 /* on vérifie que l'ouverture est correcte */
 if ( myRefNum < 0 ) return false ;

 /* selon les besoins du client :
       installation d'une alarme de réception */
 MidiSetRcvAlarm (myRefNum, myRcvAlarm) ;
 /*    installation d'une alarme d'application */
 MidiSetApplAlarm (myRefNum, myApplAlarm) ;

 /* le cas échéant, on crée un nouveau filtre */
 myFilter = MidiNewFilter () ;
 if (myFilter) {
       /* on le configure (exemple de configuration ci-
dessous) */
       SetupFilter (myFilter) ;
       /* et on l'installe */
       MidiSetFilter (myRefNum, myFilter) ;
```

```
  }

  /* on établit pour finir les connexions en entrée et en
sortie
     ici avec le pseudo-client MidiShare (numéro de
référence 0) */
  MidiConnect (myRefNum, 0, true) ;
  MidiConnect (0, myRefNum, true) ;
  return true ;
}
```

Exemple 5.2.

La fermeture d'une session consiste à libérer les ressources qui ont pu être allouées dans le cours du programme et à fermer la session MidiShare (exemple 5.3).

```
void QuitMidi()
{
  /* on libère les ressources qui ont été allouées :
     filtres, séquences etc… */
  MidiFreeFilter (myFilter) ;
  /* et on referme la session MidiShare */
  MidiClose (myRefNum) ;
}
```

Exemple 5.3.

Pour former un tout pouvant être compilé et exécuté, les exemples qui suivent présupposent l'utilisation de ce squelette. Les fonctions correspondantes viennent en complément ou en remplacement de fonctions existantes. Afin d'éviter des erreurs de compilation, il sera généralement nécessaire des les réordonner pour que leur déclaration précède leur utilisation. En utilisant un compilateur tel que gcc, vous obtiendrez un fichier exécutable avec la ligne de commande suivante :

```
# l'option -l lie le programme à la librairie MidiShare
> gcc monfichier.c -lMidiShare -o monprogramme
```

5.3.2. *Un écho*

Nous considérons ici que l'écho d'une note est une suite de notes de même hauteur, d'intensité décroissante, plus ou moins espacées dans le temps. Une application, qui génère un écho des notes reçues sur son entrée, a besoin d'installer un

filtre d'événements et une alarme de réception. Il faudra qu'elle dispose également d'une tâche temps réel pour la génération de l'écho. L'application sera paramétrée par deux variables globales : le délai entre deux échos et l'amortissement.

Le filtre de l'application permet de garantir au client MidiShare qu'il ne recevra que des événements de type Note ou KeyOn (exemple 5.4). Un événement de type Note regroupe un KeyOn et un KeyOff, il comprend un champ de durée.

```
void SetupFilter (MidiFilterPtr    filter)
{
 short i ;

 for (i=0 ;i<256 ;i++) {
        /* on rejette tous les type d'événements */
        MidiAcceptType (filter,i,false) ;
        /* on accepte tous les ports */
        MidiAcceptPort (filter,i,true) ;
 }
 for (i=0 ;i<16 ;i++)  /* on accepte tous les canaux MIDI
*/
        MidiAcceptChan (filter,i,true) ;

 /* et on accepte les Notes et les KeyOn */
 MidiAcceptType (filter, typeNote, true) ;
 MidiAcceptType (filter, typeKeyOn, true) ;
}
```

Exemple 5.4.

L'alarme de réception traite les événements reçus en temps réel. Pour chaque événement, elle installe une tâche temps réel qui est en charge de la génération de l'écho. A noter que les événements KeyOn sont transformés en notes auxquelles on affecte une durée fixe.

```
unsigned short delay=100 ;       /* les échos sont
séparés de 100 ms */
unsigned short amortissement=1 ; /* amortissement de la
vélocité */

void myRcvAlarm ( short refnum)
{
 MidiEvPtr ev ;
 while (ev = MidiGetEv (refnum)) {
        /* si l'événement n'est pas une Note c'est un
KeyOn */
```

```
        if ( EvType(ev) != typeNote ) {
              /* il est transformé en Note */
              EvType(ev) = typeNote ;
              /* on lui affecte une durée de 100 */
              Dur(ev) = 100 ;
        }

        /* un KeyOn avec une vélocité de 0 est souvent
utilisé
           à la place d'un KeyOff, d'où cette
vérification */
        if ( Vel(ev) > 0 )
              /* on programme la tâche qui génère l'écho
pour la date de
                 l'événement + le délai courant
                 l'événement à répéter est passé en
paramètre de la tâche*/
              MidiTask (EchoTask, Date(ev) + delay,
refnum, (long)ev, 0,0) ;
        else MidiFreeEv(ev) ;
  }
}
```

Exemple 5.5.

La tâche qui génère l'écho vérifie qu'il reste des échos à jouer, le cas échéant, elle en joue un et se re-programme pour la date de l'écho suivant.

```
void EchoTask (long date, short refnum, long e, long,
long )
{
 /* on récupère l'événement passé en argument */
 MidiEvPtr ev = (MidiEvPtr)e ;
 /* on calcule la nouvelle vélocité de l'événement */
 short v = Vel(ev) - amortissement ;

 /* s'il reste des échos à jouer */
 if ( v > 0 ) {
       /* on met l'événement à jour */
       Vel(ev) = v ;
       /* on émet une copie de l'événement */
       MidiSendIm (refnum, MidiCopyEv(ev)) ;
       /* et on re-programme la tâche  */
       MidiTask (EchoTask, date + delay, refnum, e, 0,
0) ;
```

```
  }
  /* on libère l'événement s'il ne reste plus d'échos à
jouer */
  else MidiFreeEv(ev) ;
}
```

Exemple 5.6.

5.3.3. *Un séquenceur*

Le séquenceur simple, que nous présentons, est une application qui peut enregistrer les événements qui se présentent sur son entrée et qui peut les relire à la demande. L'application va stocker ces événements dans une séquence MidiShare qui doit être allouée à l'initialisation et libérée à la fermeture de la session. Elle gère également un état qui permet de contrôler l'enregistrement ou la lecture. L'enregistrement se fait dans l'alarme de réception (exemple 5.7) et la lecture nécessite une tâche temps réel spécifique.

```
enum { idle, playing, recording } ;

MidiSeqPtr mySeq ;
int state = idle ;
void myRcvAlarm (short refnum)
{
  /* si le séquenceur est en enregistrement */
  if (state == recording) {
        MidiEvPtr ev ;
        /* pour chaque événement reçu */
        while (ev = MidiGetEv(refnum)) {
              /* envoie une copie de l'événement
                 (fonctionnalité de thru optionnelle) */
              MidiSendIm (refnum, MidiCopyEv(ev)) ;
              /* et stocke l'événement dans la séquence
MidiShare */
              MidiAddSeq (mySeq, ev) ;
        }
  }
  /* le séquenceur n'enregistre pas : on libère tous les
événements */
  else MidiFlushEvs (refnum) ;
}
```

Exemple 5.7.

La tâche qui relit une séquence enregistrée est initialisée avec le premier événement de la séquence. Son principe est le suivant : elle joue tous les événements qui sont à la même date. Quand elle rencontre un événement qui n'est plus à la date courante, elle se re-programme pour la date correspondant à cet événement en le passant comme argument du prochain appel (exemple 5.8).

A titre de rappel, les événements sont ordonnés par dates croissantes dans les séquences MidiShare.

```
void PlayTask (long date, short refnum, long e, long a,
long b)
{
 /* on récupère l'événement courant */
 MidiEvPtr ev = (MidiEvPtr)e ;
 /* on préserve sa date */
 long offset, d = Date(ev) ;

 /* on vérifie que le séquenceur joue */
 if (state != playing) return ;

 /* pour chaque événement dans la séquence */
 while (ev) {
        /* on calcule son offset avec le premier
événement joué */
        offset = Date(ev) - d ;
        /* sa date est différente : on sort de la boucle
*/
        if (offset) break ;
        /* on envoie une copie de l'événement */
        MidiSendIm (refnum, MidiCopyEv (ev)) ;
        /* et on passe à l'événement suivant */
        ev = Link(ev) ;                    // go to next
event
 }
 /* s'il reste des événements à jouer, on programme la
tâche
    pour le futur avec l'événement courant comme argument
*/
 if (ev) MidiTask (PlayTask, date + offset, refnum,
(long)ev, 0, 0) ;
 else state = idle ;
}
```

Exemple 5.8.

De manière rudimentaire, le corps du programme peut automatiquement enregistrer au démarrage, puis rejouer ensuite la séquence enregistrée (exemple 5.9) avant de quitter.

```
/* allocation d'une nouvelle séquence */
mySeq = MidiNewSeq () ;
/* changement d'état pour enregistrer */
state = recording ;

printf ("appuyer sur <return> pour arrêter
l'enregistrement") ;
getc(stdin) ;
/* arrêt de l'enregistrement */
state = idle ;

printf ("appuyer sur <return> pour commencer la
lecture") ;
getc(stdin) ;
/* activation de la tâche de lecture (si la séquence
n'est pas vide) */
if (First(mySeq)) {
 state = playing ;
 PlayTask (MidiGetTime(), myRefNum, (long)First(mySeq),
0, 0) ;
}
/* attente de la fin de la lecture */
while (state == playing)
 sleep (1) ;
```

Exemple 5.9.

5.4. Quelques difficultés de la programmation temps réel

Si l'utilisation de MidiShare permet une écriture simple et concise des tâches temps réel, un certain nombre de problèmes restent à résoudre quand on va dans le détail des implémentations. Parmi eux figurent notamment les problèmes de latence, de précision et de dérive temporelle.

5.4.1. *Composer avec la latence*

Nous désignons ici la *latence* comme étant le délai qui sépare la date d'échéance d'une tâche de sa réalisation effective. Ces délais peuvent intervenir à tous les niveaux dans la chaîne de traitement d'un message musical : quand l'ordonnanceur

du système d'exploitation est en jeu (particulièrement dans le cas des langages non temps réel tels que Lisp ou Java) ou encore quand les événements sont délivrés *via* un canal de communication dont les temps de transmission ne sont pas déterministes (par exemple : cas des transmissions sur un réseau). Des techniques particulières doivent alors être utilisées pour compenser la latence et conserver strictement l'ordonnancement temporel des événements qui sont transmis.

Nous allons supposer que nous sommes dans le cas d'un langage non temps réel tel que Java. Nous avons programmé une tâche qui émet un événement à intervalles réguliers. Cette tâche se re-programme elle-même avec la périodicité correspondante. Cependant, à sa date d'échéance, l'activation de la tâche prendra un temps variable, dépendant essentiellement du contexte courant de la machine. Ce temps est généralement borné et nous désignons par *maxlat*, la valeur maximale de ce délai. Pour une périodicité souhaitée *P*, la périodicité réelle de chaque tâche va donc varier en fonction de la latence courante et en conséquence, l'ordonnancement temporel des événements émis sera déformé. Une solution consiste à anticiper la variation de la latence et à décaler les événements dans le futur d'une valeur qui soit au moins égale à *maxlat*, de manière à s'assurer que ces événements ont bien été émis au moment de leur date d'échéance (exemple 5.10).

```
/* la tâche prend comme argument l'événement à émettre,
la période
   la latence maximale attendue */
void myTask (long date, short refnum, long e, long
periode, long maxlat)
{
 /* on envoie une copie de l'événement dans le futur,
décalé de la
    valeur de la latence maximale */
 MidiSendAt (refnum, MidiCopyEv((MidiEvPtr)e), date +
maxlat) ;
 MidiTask (myTask, date + periode, refnum, e, periode,
maxlat) ;
}
```

Exemple 5.10.

Cette technique introduit généralement un délai faible, tolérable dans la plupart des contextes musicaux, ce qui la rend applicable dans bien des cas. Elle est notamment mise en œuvre pour compenser les délais de transmission d'événements musicaux sur Internet [FOB 01].

5.4.2. *Dérive temporelle*

Typiquement, les dérives temporelles affectent les tâches qui se répètent elles-mêmes en faisant usage d'un offset pour calculer la date d'échéance suivante. En effet, la date passée en argument de la tâche (sa date d'échéance) ne correspond pas toujours à la date courante du système. Cela peut provenir des effets de la latence ou encore être le fait d'une tâche qui a été programmée *en retard*.

Reprenons l'exemple de la tâche périodique précédente : si cette tâche faisait usage de la date courante du système pour se reprogrammer, elle dériverait dans le temps en fonction des variations de la latence (exemple 5.11).

La solution consiste ici à faire rigoureusement usage de la date *logique* de la tâche (voir exemple 5.10). La même règle s'applique pour les événements.

```
void myTask (long date, short refnum, long periode, long,
long)
{
 /* la tâche fait usage de la date du système et n'est
donc pas à l'abri de dérives temporelles */
 MidiTask (myTask, MidiGetTime() + periode, refnum,
periode, 0, 0) ;
}
```

Exemple 5.11.

Les approximations de calcul sur les dates constituent également une cause importante de dérive temporelle. Ces approximations interviennent généralement quand il s'agit de convertir une date entre différentes références de temps : du temps musical vers le temps absolu par exemple. Le temps musical fait usage d'unités qui dépendent du tempo courant. Ce tempo peut être exprimé de différente manière : il est représenté en microsecondes par noire dans la norme MIDI File, il consiste en nombre de battues par minute pour les musiciens.

Quelque soit la précision des nombres utilisés pour calculer les dates, les divisions par exemple, produiront généralement des approximations qui cumulées, conduiront à terme à une dérive temporelle. La solution consiste ici à conserver les valeurs ignorées et à les réinjecter dans le calcul quand elles deviennent significatives (exemple 5.12).

```
long long tempo ; /* exprimé en microsecondes par noire
*/
                  /* un tick représente 1/24 de noire */
                  /* le type 'long long' permet d'éviter
*/
                  /* les débordements lors des calculs */
void myTask (long date, short refnum, long ticks, long,
long)
{
 static long reste = 0 ;
 /* conversion en temps absolu (millisecondes) */
 long long offset = (tempo * ticks) / 24000 ;
 /* on garde le reste de la conversion */
 reste += (long)(tempo * ticks) % 24000 ;
 /* et au besoin, on le réinjecte dans l'offset */
 while (reste > 1000) {
       offset += 1 ;
       reste -= 1000 ;
 }
 MidiTask (myTask, date + offset, refnum, ticks, 0, 0) ;
}
```

Exemple 5.12.

5.5. Conclusion

Le développement d'applications musicales requiert une maîtrise du temps qui suppose à la fois une bonne connaissance et une grande pratique des couches les plus basses des systèmes d'exploitation ciblés. Nous avons vu avec MidiShare, que la mise en œuvre d'une architecture logicielle dédiée aux types de problèmes que soulèvent les applications musicales, permet de simplifier le travail du développeur de manière très significative, en fournissant notamment des fonctions de haut niveau

pour la gestion du temps ainsi qu'une représentation structurée des événements musicaux. MidiShare est un logiciel « Open Source », librement accessible à l'adresse suivante : http://www.grame.fr/MidiShare.

Nous avons également vu que l'utilisation de cette architecture ne résout pas tous les problèmes, quelques-uns parmi ceux qui relèvent toujours de la gestion du temps ont été abordés. Ce n'est qu'à travers une bonne connaissance du système, alliée à une expérience assidue que l'on parvient à maîtriser les difficultés de la programmation temps réel. Nous ne saurions trop conseiller aux développeurs de soigner leur code avec la minutie d'un horloger.

5.6. Bibliographie

[FOB 01] FOBER D., LETZ S., ORLAREY Y., « Real Time Musical Events Streaming over Internet », *Proceedings of the International Conference on WEB Delivering of Music, IEEE*, p. 147-154, 2001.

[FOB 02] FOBER D., LETZ S., ORLAREY Y., « Lock-Free Techniques for Concurrent Access to Shared Objects », *Actes des Journées d'Informatique Musicale JIM2002, GMEM,* Marseille, p. 143-150, 2002.

[MM 90] MIDI Management Tools, Apple Computer, Inc., 1990.

[OMS 95] OMS (Open Music System) Programming Interface, Opcode System, Inc, 1995.

[ORL 89] ORLAREY Y., LEQUAY H., « MidiShare: a Real Time multi-tasks software module for Midi applications », *Proceedings of the ICMC, ICMA*, San Francisco, p. 234-237, 1989.

[ORL 90] ORLAREY Y., « An Efficient Scheduling Algorithm for Real-Time Musical Systems », *Proceedings of the ICMC, ICMA*, San Francisco, p. 194-198, 1990.