UNIVERSITÉ DE
SHERBROOKE

IFT 729
Conception de systèmes temps-réel
Livrable 02

Mon piano d'amour

JONATHAN ACEITUNO
Matricule 09187925
GAFAR AMINOU
Matricule 06794089

Sommaire

1 Introduction **4**
- 1 **Promesses tenues** . . . 4
- 2 **Changements par rapport aux technologies utilisées** . . . 5
- 3 **Utilisation du livrable** . . . 5
  - 1 Windows . . . 5
  - 2 UNIX . . . 5
  - 3 Mac OS X . . . 5

2 Développement multiplateforme et contraintes temps-réel **7**
- 1 **Environnement de développement** . . . 7
- 2 **Modification des contraintes temps-réel** . . . 8

3 Architecture du piano d'amour **10**
- 1 **Classes communes** . . . 10
  - 1 Notion de sous-système . . . 10
  - 2 Gestion des erreurs . . . 11
  - 3 Représentations des touches du piano . . . 11
  - 4 File bornée . . . 12
  - 5 Moyenneur . . . 12
- 2 **Multiprogrammation** . . . 12
  - 1 Thread . . . 13
  - 2 Mutex, autoverrou, semaphore, condition . . . 13
  - 3 Tampon partagé . . . 13
- 3 **Interface** . . . 14
  - 1 Module d'interface . . . 14
  - 2 Surfaces graphiques . . . 15
  - 3 Gestion des événements . . . 15
  - 4 Éléments d'interaction . . . 15
- 4 **Réseau** . . . 15
- 5 **Audio** . . . 16
  - 1 Son . . . 16
  - 2 Événement audio . . . 16
  - 3 Lecture du son . . . 17

4 Chargeur de notes . . . 17
5 Dispatcheur audio . . . 17
6 **Gestion du temps** . . . 17
1 Abstraction des opérations du temps . . . 18
2 Chronomètre d'athlétisme . . . 18
3 Minuterie de cuisine . . . 19
4 Cycle . . . 19
5 Docteur Emmett Brown . . . 19
7 **Exécution du programme** . . . 19
1 Classe principale . . . 19
2 Cueilleur de touches . . . 20
3 Client . . . 20
4 Serveur . . . 20

4 Réalisation **22**
1 **Protocole de connexion** . . . 22
2 **Mécanisme de synchronisation** . . . 22
3 **Gestion de l'exécution des tâches** . . . 23
4 **Audio** . . . 23
1 Préparation des échantillons audio . . . 23
2 Chargement des échantillons en mémoire . . . 25
3 Choix du format d'échantillonnage . . . 25
a Taille du tampon . . . 25
b Format des échantillons . . . 26

5 Tests et calibration **28**
1 **Partie audio** . . . 28
1 Motivations . . . 28
2 Méthodologie de test . . . 29
3 Résultats des tests . . . 29
4 Prises de décision . . . 30
2 **Partie réseau** . . . 30
1 Motivations . . . 30
2 Méthodologie de test . . . 31
3 Résultats des tests . . . 31
4 Prises de décision . . . 31
3 **Mise en situation réelle** . . . 32

6 Conclusion **33**

CHAPITRE 1

Introduction

Le **livrable 02** rend compte de la version finale de l'application Mon piano d'amour. Comme ce fut le cas pour le livrable 01, certaines prédictions ont été confirmées et d'autres ne sont plus valables. Cependant, nous avons tenu à fixer certains acquis du livrable 01, ou du moins à les améliorer. Nous tenons pour acquis les propos qui ont été tenus dans les précédents livrables.

SECTION

Promesses tenues

Le livrable 00 décrivait ce présent livrable comme la version termiinée de l'application. Il s'agissait donc de l'application à date du livrable 01, augmentée :

– de la possibilité de pouvoir jouer toutes les notes d'une tessiture relativement importante ;
– de la capacité à jouer plusieurs sons en même temps ;
– de moduler les notes jouées par au moins un effet *pédale douce* [1] ,
– d'une certaine garantie de respect des contraintes TR fixées.

On peut donc raisonnablement qualifier le système courant de système temps réel au sens usuel, mais uniquement en reconnaissant qu'il y eut non seulement un changement dans les contraintes fixées, au vu du constat de contraintes techniques sur l'environnement d'exécution du programme, mais aussi en considérant une certaine marge d'erreur attribuée à l'incertitude quant aux éléments extérieurs influant sur le système. Ces thèmes seront détaillés plus loin dans le rapport.

[1] La pédale est assignée à la touche Shift gauche, et les sons joués avec cette pédale seront atténués.

SECTION 2

Changements par rapport aux technologies utilisées

Nous avons choisi de ne pas changer les technologies utilisées, faisant reposer notre code sur la bibliothèque multi-plateforme **SDL** et ses modules **SDL_ttf**, **SDL_image** et **SDL_net**. En plus de cette base déjà confirmée au livrable 01, nous avons accueilli **SDL_mixer** qui est une simple console de mixage à utiliser avec le système audio de **SDL**. Son évaluation et sa critique sera faite plus loin. Lors de nos tests, nous nous sommes également confrontés à la bibliothèque **Portaudio**, plus adaptée aux logiciels audionumériques, mais les contraintes liées à l'exécution du programme final ont fait ressortir que ce choix plus adapté nécessitait des prérequis inacceptables pour utiliser le programme.

SECTION 3

Utilisation du livrable

Il est possible d'obtenir les sources et les exécutables du livrable pour les différentes plateformes à l'adresse `http://oin.name/divers/monpianodamour`.

❶ Windows

Exécutable Lancer l'exécutable.
Sources Projet dans `win32\`. Requiert NetBeans 6.8 et Mingw.

❷ UNIX

```
cd linux/
make
./monpianodamour
```

Requiert les librairies SDL.

❸ Mac OS X

Exécutable Lancer l'application dans l'image disque
Sources Projet dans `macosx/`. Requiert Xcode et les frameworks SDL.

Au démarrage, le programme demande à choisir la personnalité qu'il adoptera, entre client et serveur. Le serveur s'assurera d'entrer la latence voulue et d'allumer ses haut-parleurs avant de démarrer. Le client, une fois connecté, utilise les touches du clavier d'ordinateur comme indiqué sur l'interface pour jouer du piano. Un bouton permet de changer entre clavier QWERTY et AZERTY. Les touches **W**, **X**, **C** et **V** permettent de choisir la hauteur d'octave actuelle des notes qu'on peut jouer sur le clavier. Le piano d'amour n'offre pas de mécanisme d'adieu lors d'une déconnexion, il est donc de votre responsabilité de fermer le client et le serveur.

CHAPITRE 2

Développement multiplateforme et contraintes temps-réel

Les différentes contraintes que nous avons énoncé au tout début du projet ont été adaptées au contexte défini à la fois par notre environnement de développement et par la diversité des plateformes sous lesquelles notre projet devait s'exécuter.

SECTION 1

Environnement de développement

Nous avons souffert de la configuration particulière de notre environnement de développement. Le produit fini devait être disponible pour la plateforme Win32 pour pouvoir subir la correction, or nos deux ordinateurs hébergeaient le système d'exploitation Mac OS X et nous n'avions pas d'accès administrateur à un ordinateur sous Windows. Il nous aurait été possible de faire une requête pour en obtenir un si jamais cette nécéssité avait été établie dès le début, mais nous avons choisi de continuer en privilégiant le fait que le code devait être multiplateforme.

Nous avons utilisé l'environnement de développement intégré Xcode[1] pour le développement sur notre plateforme d'origine, et nous avons utilisé le logiciel de gestion de versions Subversion pour synchroniser nos travaux et y accéder sur une base commune.

Lorsque nous avons voulu faire en sorte que le programme compile avec d'autres plateformes, nous avons cherché à utiliser des outils proches de ceux qui étaient offerts par notre plateforme d'origine, et c'est pourquoi nous utilisons Mingw pour Windows et GNU gcc pour les autres UNIX.

Nous avons paramétré un projet NetBeans pour assurer l'édition et la compilation sous Windows, et nous avons rédigé un `Makefile` pour les systèmes proches d'UNIX.

Trois ordinateurs portables ont été utilisés pour nos développements :

---

[1] C'est l'EDI *de facto* sur la plateforme Mac OS X puisqu'il est fourni gratuitement avec chaque ordinateur et disponible en téléchargement. Xcode utilise la collection de compilateurs GNU gcc et le débogueur GNU gdb, dans des versions adaptées pour supporter les spécificités de la plateforme, mais ces adaptations ne concernent que le langage Objective-C.

- Macbook Pro late 2009 à processeur Core 2 Duo (2,53 Ghz), 4 Go de mémoire vive, Mac OS 10.6.2 / 10.6.3
- Macbook noir 2 Ghz à processeur Core 2 Duo (2 Ghz), 2 Go de mémoire vive, Mac OS 10.5.8
- Lenovo 3000 C100 à processeur Pentium M (1,7 Ghz), 512 Mo de mémoire vive, Ubuntu 9.10 avec noyau Linux 2.6.31

Nous avons également utilisé deux machines virtuelles émulant les systèmes suivants, pour la mise en place des versions multiplateforme :
- une hébergeant Windows XP SP3 ;
- et une hébergeant Ubuntu 9.10 avec le noyau 2.6.31.

SECTION 2 Modification des contraintes temps-réel

Les raisons des changements suivants seront abordés plus loin dans le rapport. Lorsque nous avons établi le cahier des charges du projet, au livrable 00, nous avions défini les contraintes suivantes :
- une touche tapée à destination du programme doit toujours être lue, ce qui peut se traduire par une **contrainte d'immédiateté**, dans le sens où une touche pressée doit être connue assez rapidement avant que l'arrivée d'une nouvelle touche ne puisse masquer cette précédente information ;
- le délai entre l'action de taper une touche reconnue comme une note par le programme client et la lecture du son dans les haut-parleurs de l'ordinateur exécutant le programme serveur, soit le délai entre le début du traitement et sa fin, est constant et fixé à 25ms, lorsque les conditions réseau sont clémentes, ce qui peut être considéré comme une sorte de **contrainte de brièveté**, ou le temps minimum et le temps maximum toléré pour réaliser l'opération convergent ;
- la préparation d'un son donné doit être suivie de la sorte dans les haut-parleurs dans un délai maximum de 10ms, ce qui fait une **contrainte** locale **d'immédiateté** ;
- dans la mesure où le lien réseau utilisé offre une latence de 10ms, le module réseau devrait assurer une latence de 15ms, ce qui est relativement imprécis, puisqu'on ne dit pas quelle partie du traitement du réseau est concernée ;

À la lumière de nos observations, qui seront détaillées ultérieurement, il apparaît que nous ne pouvions pas rencontrer la contrainte de brièveté décrite ci-dessous, ce qui est un handicap majeur pour ce projet dont les objectifs étaient avant tout d'être multiplateforme et de pouvoir accuser une latence assez petite pour permettre un jeu confortable.

Nous avons déterminé que la résolution de ce problème demandait d'exiger de l'utilisateur des prérequis très spécifiques aux différentes plateformes, et quelquefois l'installation de pilotes de carte son à basse latence. Cela nous a paru inacceptable, et de plus nous n'avons pas eu accès à un ordinateur sous Windows de façon à pouvoir modifier ses pilotes, ce qui nous limitait sérieusement pour les tests. Si nous voulions continuer à proposer une latence la plus basse possible avec n'importe quel système, il nous aurait fallu être

plus flous sur les contraintes temps réel, ce qui n'aurait plus vraiment eu de sens puisque le mot d'ordre aurait été de *faire tout le plus vite possible et croiser les doigts*, ce qui ne nous aurait pas vraiment appris du cours.

Nous avons donc décidé de modifier les paramètres de la contrainte de manière à ce que la latence de la carte son, imprévisible car dépendant des pilotes, soit négligeable par rapport au temps total du traitement qui caractérise la contrainte de brièveté. Ce temps doit pouvoir être choisi par l'utilisateur au démarrage du programme.

Les contraintes de temps réel auxquelles notre programme doit être soumis sont donc les suivantes :

- le temps total depuis l'appui sur une touche significative[2] sur le poste client jusqu'à la lecture effective du son sur la carte son par le poste serveur est réglable à l'exécution et ne doit pas être dépassé ; c'est cette opération au complet qui est donc l'objet d'une contrainte de brièveté ;
- la préparation d'un son donné doit être suivie de la sortie dans les haut-parleurs dans un délai minimal ;
- l'envoi et la réception des informations par le réseau doit être assez fiable pour qu'aucune note ne soit oubliée.

En un sens, la première de ces contraintes est la plus importante et se trouve répartie sur plusieurs modules et même à travers le réseau. Son respect découle donc d'une multitude de décisions que nous avons prises afin de le garantir dans des circonstances normales de fonctionnement.

[2] On appelle touche significative une touche qui correspond à une note du piano.

CHAPITRE 3

Architecture du piano d'amour

Le piano d'amour a subi des changements de conception aussi souvent qu'il a été nécessaire pour nous permettre d'être à l'aise avec notre réalisation. Si nous avons perdu beaucoup de temps sur ce point, nous avons par contre mis au point une conception modulaire où les modules peuvent servir indépendamment pour d'autres buts. Nous proposons donc pour ce livrable de faire table rase du passé et nous présenterons donc l'intégralité de l'architecture, même si certains éléments peuvent paraître familiers au lecteur assidu du livrable 01. Nous utilisons intensivement la librairie SDL et ses extensions SDL_image, SDL_ttf, SDL_net et SDL_mixer pour implémenter tous les modules, et cela permet d'assurer un fonctionnement similaire dans toutes les plateformes.

SECTION 1 Classes communes

Nous avons conçu ce module comme un fourre-tout où les choses qui pouvaient être utiles à toutes les parties seraient entreposées. Il s'agit donc de quelques classes soit utilitaires, soit générales.

1 Notion de sous-système

La notion de sous-système a déjà été introduite dans le livrable 01. Il s'agit d'un *presque singleton* dans le sens où la sémantique est un peu différente : ici, un sous-système est instancié et sa durée de vie est entre les mains du responsable de son instanciation, et des objets peuvent y faire appel pendant tout le temps de sa durée de vie, mais ni avant ni après. Afin de s'assurer qu'un certain objet serait instancié uniquement lorsqu'un sous-système dont il dépend est déjà lancé, nous avons forgé le patron de classe `dependant_de<T>`. Cela nous permet de faciliter la création sécuritaire d'objets RAII.

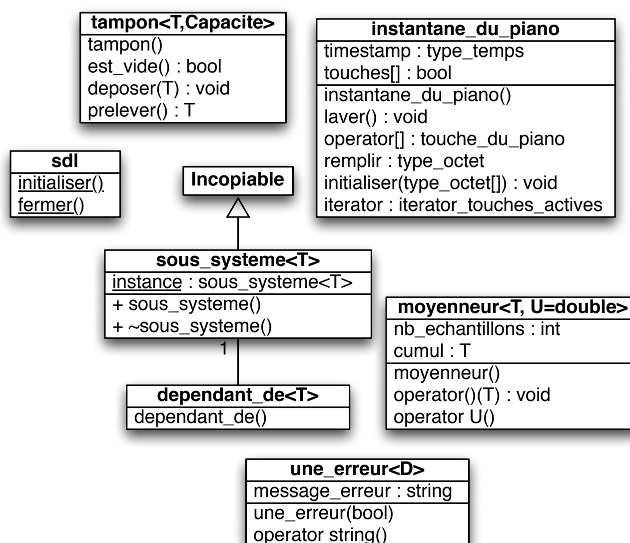

Figure 1.1 Classes communes

❷ Gestion des erreurs

Notre programme utilise les exceptions du C++ avec un type d'exception particulier, `erreur`, et la règle ``jeter par valeur, rattraper par référence'' pour le traitement des exceptions. Le programme déléguant souvent à la bibliothèque SDL la description des détails d'une erreur, et les types d'erreurs étant souvent les mêmes, nous avons créé le patron de classe `une_erreur<D>`, avec `D` une chaîne de caractères C. La création d'une nouvelle erreur consiste ainsi uniquement en la déclaration puis la définition d'une chaîne de caractères C constante, soit comme constante globale, ce qui fut notre choix, soit comme constante placée dans un endroit judicieusement choisi.

❸ Représentations des touches du piano

Le programme utilise deux façons de représenter les touches du piano, que l'on peut rapprocher des concepts de représentation parallèle ou en série. Ici, la représentation *en série* est un simple `enum` recensant toutes les notes qu'il est possible de trouver sur un piano, ce qui permet de décrire plusieurs notes comme une série de valeurs de cet `enum`, tandis que la représentation *en parallèle* est une classe complète, la classe `instantane_du_piano`, qui comme son nom l'indique représente la totalité des états du piano à un instant donné.

Nous avons mis au point cette double représentation afin de répondre correctement aux exigences de fiabilité et de rapidité que nous avions évoquées au livrable 00. L'envoi séquentiel de notes par le réseau une fois qu'elles sont tapées présente le double risque de les perdre et de mélanger leur ordre d'arrivée lorsque l'envoi est fait avec UDP. Ces risques sont réduits à néant en utilisant le protocole de transport TCP, mais au prix d'un surcoût

trop important qui comprometterait toute tentative de détermination d'un temps de transmission[1] : il est clair que nous avons besoin de contrôler le mécanisme de transmission pour rencontrer les contraintes fixées. Ainsi, pour éviter tout de même les pertes qui sont inacceptables du point de vue des contraintes, nous utilisons l'approche adoptée avec certains jeux en réseau : une connexion TCP est disponible pour les informations non urgentes et dont la bonne arrivée est cruciale, et l'état du jeu est transmis sur UDP sur une base périodique, avec des instantanés. Nous faisons donc deux changements de représentation dans tout le programme. Tout d'abord, la représentation en série est de rigueur puisque l'interface graphique fonctionne par événements et non par sondage. Ensuite, cette représentation en série alimente une représentation en parallèle qui est envoyée sur une base périodique par le réseau. De là, le serveur peut analyser chaque instantané, prendre uniquement ceux qui sont pertinents, et transformer à nouveau ces instantanés en événements qui seront déclenchés au bon moment et qui donneront naissance à des sons. De cette manière, la redondance des informations garantit l'arrivée de chaque note, si tant est que la connexion est assez solide.

❹ File bornée

Le patron de classe `tampon<T,Capacite>` représente une file à capacité limitée d'objets de type `T`. La raison de préférer d'utiliser ceci plutôt qu'un conteneur STL est inexistante, et cette classe n'est pas utilisée pour elle-même actuellement.

❺ Moyenneur

Nous avons mis à disposition `moyenneur<T,U>`, un foncteur automatisant la notion de moyenne de type `U` sur un ensemble d'échantillons d'un type `T`. Chaque fois que le foncteur est appelé avec un argument de type `T`, un échantillon supplémentaire est ajouté au cumul des valeurs de tous les échantillons, puis lorsqu'on le désire on peut demander la moyenne arithmétique des échantillons.

SECTION 2 Multiprogrammation

Bien que notre utilisation de la multiprogrammation se limite à quelques sous-programmes de tests, nous avons envisagé la possibilité de l'utiliser pour gérer l'accomplissement des différentes tâches du système, et c'est pourquoi nous avions besoin d'objets prêts à l'emploi et abusant de l'idiome RAII pour développer rapidement et de façon sécuritaire de multiples threads. L'implémentation sous-jacente et tout le travail difficile de différenciation multi-plateforme a déjà été fait dans la librairie SDL.

[1] En effet, avec TCP lorsqu'un paquet est perdu, il faut attendre qu'il soit réémis.

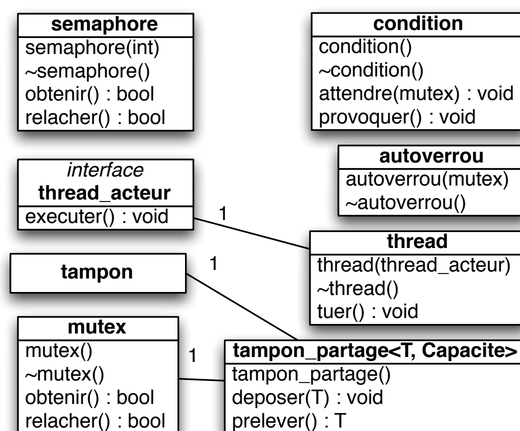

Figure 2.1 Classes utiles en situation de multiprogrammation

❶ Thread

Une instance de `thread` est un objet RAII qui automatise la création d'un thread à sa construction et l'attente *à la join* du thread à sa destruction. C'est une instance d'un enfant de la classe `thread_acteur` qui contient une méhode `executer()` qui sera appelée au lancement du nouveau fil d'exécution.

❷ Mutex, autoverrou, semaphore, condition

Nous avons repris les idées géniales du prof en ce qui concerne le verrouillage RAII. Les mutexes, sémaphores et conditions[❷] sont des classes RAII également.

❸ Tampon partagé

Nous avons encapsulé l'utilisation d'un `tampon` des classes communes de manière protégée, en répondant au problème des producteurs et des consommateurs grâce à des mutexes et des conditions. Ce type de tampon nous aurait servi afin de garder des tampons *thread-safe* pour le passage de représentations *série/parallèle* pour les notes du piano, du côté client comme du côté serveur. Nous avions même cherché une alternative plus efficace du côté des implémentations *lock-free* et avons étudié l'implémentation utilisée dans la librairie Midishare[❸] puis avons capitulé face à tant d'assembleur et de *compare-and-swap*.

---

❷ Ce sont des conditions style POSIX, mais utilisables également sous Windows.

❸ Une librairie C crée par le centre de recherche GRAME (Lyon, France) pour le transport d'événements MIDI par le réseau à basse latence, en quelque sorte une version sérieuse de notre piano d'amour, sans les sons !

SECTION 3

Interface

Nous avons abandonné l'abstraction un peu trop indigeste que nous avions monté pour le livrable 01 au sujet de l'interface, au profit d'un modèle plus simple et plus efficace.

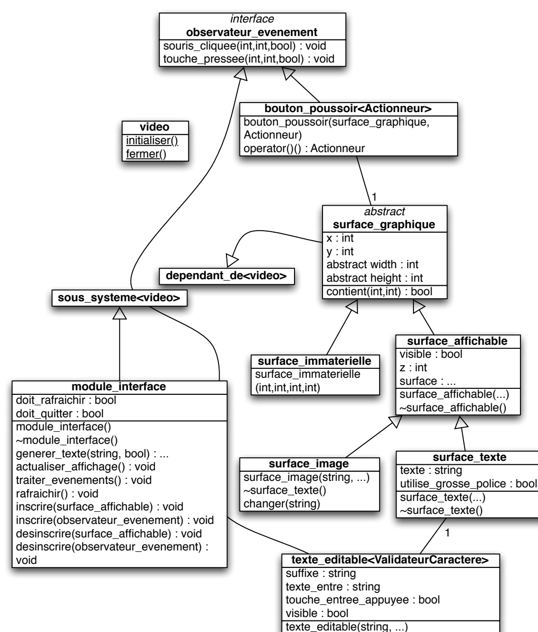

Figure 3.1 Classes de l'interface

❶ Module d'interface

Le module d'interface est un de ces sous-systèmes dont on a déjà parlé mais il possède des facettes spécifiques à la gestion d'une interface. Il automatise l'allocation et la gestion de l'écran, des polices d'écriture[4], du pipeline d'affichage et des événements grâce aux méthodes `inscrire()` et `desinscrire()` que peuvent appeler les objets d'interface durant, respectivement, leur construction et leur destruction. La stratégie d'affichage est similaire au mécanisme de *lazy copy* qui consiste à ne travailler qu'au moment où c'est vraiment nécessaire[5]. Ainsi, l'affichage n'est pas réactualisé tant qu'un objet graphique n'a pas notifié un changement auprès du module d'interface. Ce dernier, enfin, a la responsabilité de déterminer s'il faut quitter l'interface ou non.

---

[4] Actuellement il y a deux polices d'écriture : la petite maigre et la grande grosse. Les hommes préféreront sans doute la grande maigre et la petite grosse, et plus probablement encore la grande maigre à la petite grosse.

[5] Il est important de ne pas confondre avec le mécanisme de *procrastination* qui consiste à ne travailler qu'après le moment où c'est nécessaire.

❷ Surfaces graphiques

Les objets d'interface les plus primaires sont les surfaces. Une surface graphique, représentée par la hiérarchie descendant de la classe abstraite `surface_graphique`, possède un rectangle qui délimite sa position et sa taille. Il existe deux types de surfaces graphiques : les surfaces immatérielles, qui représentent uniquement des rectangles sans rien afficher, et les surfaces graphiques qui permettent d'afficher quelque chose dans le rectangle, et de gérer une information de profondeur. Deux types de surfaces affichables existent : les surfaces de type image, avec lesquelles ont peut rapidement charger des images, et les surfaces de type texte, qui utilisent une police de l'interface pour faire leur rendu.

❸ Gestion des événements

Les objets voulant être notifiés de certains événements n'ont qu'à descendre de la classe `observateur_evenement`, qui se charge d'inscrire l'objet instancié au pipeline des événements de l'interface de manière automatique. Il suffit juste d'implémenter les méthodes `souris_cliquee` et `touche_pressee` et d'agir en conséquence.

❹ Éléments d'interaction

Nous avons préparé deux éléments d'interaction, qui sont une zone d'édition de texte `texte_editable<ValidateurCaractere>` conseillée éventuellement par un foncteur pour le choix des caractères à accepter, et un bouton poussoir dont le comportement en réponse à un clic est à déterminer, bien qu'un comportement passif soit disponible par défaut.

SECTION 4 Réseau

Nous avons continué dans la lancée de nos observations du livrable 01 en ce qui concerne le réseau, et il est donc établi que la connexion entre un client et un serveur s'effectue avec une connexion TCP mais laisse l'ouverture d'un port UDP afin de permettre une communication rapide et avec pertes pour les phases sensibles.

Les classes `client_reseau` et `serveur_reseau` n'ont pas grand chose d'original puisqu'il s'agit d'établir une connexion par TCP puis de préparer des ports UDP pour une communication par un canal plus rapide mais moins fiable. L'émission et la réception de paquets sur les deux protocoles de transport sont délégués aux classes `emetteur_reseau` et `recepteur_reseau`.

Le protocole de connexion sera expliqué plus loin puisqu'il comporte quelques spécificités, notamment en ce qui concerne la préparation de la *liaison* par UDP. Le module réseau renferme également l'exécution du protocole de synchronisation servant à la communication et à l'évaluation d'un offset de temps entre le temps client et le temps serveur,

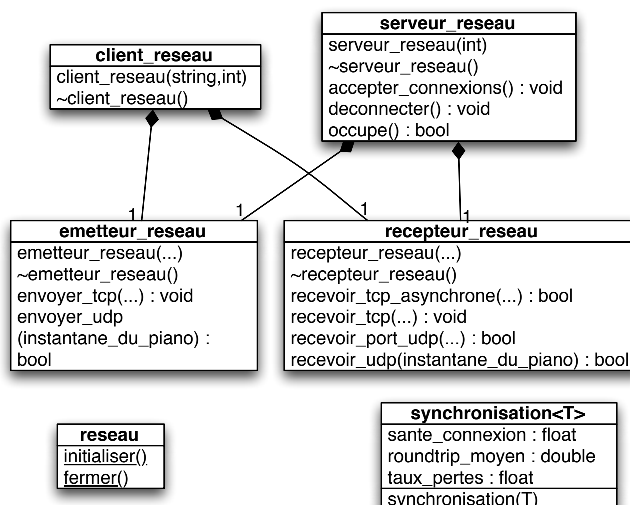

Figure 4.1 Classes pour le réseau

mais aussi à récolter des données statistiques, à l'ouverture de connexion, sur l'état de la connexion.

SECTION 5

Audio

La librairie SDL_mixer a grandement facilité le développement de la partie audio puisqu'elle permet de rajouter des fonctions de mixage simples mais pas trop au-dessus des fonctions bas niveau de son de SDL, tout en laissant un contrôle total sur les propriétés du flux audio désiré [6] .

1 Son

La classe RAII `son` charge un son à partir du chemin d'un fichier WAV, et ce en deux exemplaires : un pour le son à pleine puissance et un autre pour le son atténué. Cela permet de minimiser le temps de calcul entre le choix du son à jouer et la lecture proprement dite : aucun calcul éventuel n'est à faire puisque tout est en mémoire.

2 Événement audio

Un `evenement_audio` représente la donnée d'une note jouée à un moment donné dans le temps de manière atténuée ou non.

[6] Dans la limite de ce qu'il est possible d'obtenir, bien entendu...

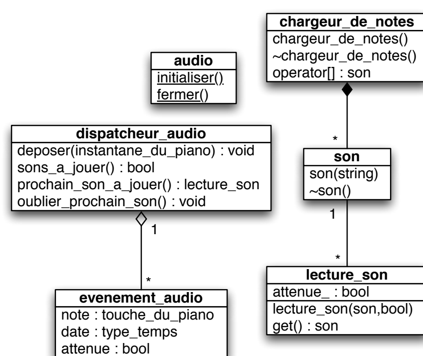

Figure 5.1 Classes gérant le son

❸ Lecture du son

La classe `lecture_son` représente un jeu de paramètres pour la lecture d'un son donné. Il n'y a qu'un seul paramètre pour l'instant, il s'agit de l'atténuation.

❹ Chargeur de notes

La classe `chargeur_de_notes` est une classe RAII chargée de charger (*insert-sic-here*) l'ensemble des sons correspondant aux notes du piano dans la mémoire vive, durant une période creuse. Elle est également chargée de fournir une méthode d'accès à ces sons en fonction de la note qu'ils représentent.

❺ Dispatcheur audio

La classe `dispatcheur_audio` est responsable de gérer l'arrivée de nouveaux instantanés du piano, de filtrer ceux qui seraient périmés, et surtout de les transformer en événements audio, donc de passer d'une représentation *en parallèle* à une représentation *en série* tout en gérant intelligemment les surplus d'information inévitables. C'est le `dispatcheur_audio` qui sera en charge de conseiller le serveur sur le bon moment pour jouer les sons qu'il contient de façon à respecter la contrainte temps-réel de brièveté en fonction de la latence voulue par l'utilisateur. Pour cela, le `dispatcheur_audio` délègue les questions de conversion de référentiel temporel au docteur Emmett Brown.

SECTION 6 Gestion du temps

À date du livrable 01, la gestion du temps était, avec la gestion du son, une des parties les moins développées de notre réalisation. Aujourd'hui le module de gestion du temps

comprend ce qu'il faut pour non seulement abstraire la représentation du temps, mais également les opérations et objets pouvant agir sur ce temps. Il est aussi question de gérer le temps selon un référentiel, puisqu'il faut que le programme puisse tirer des informations à partir d'un moment du temps donné selon le référentiel d'une machine distante.

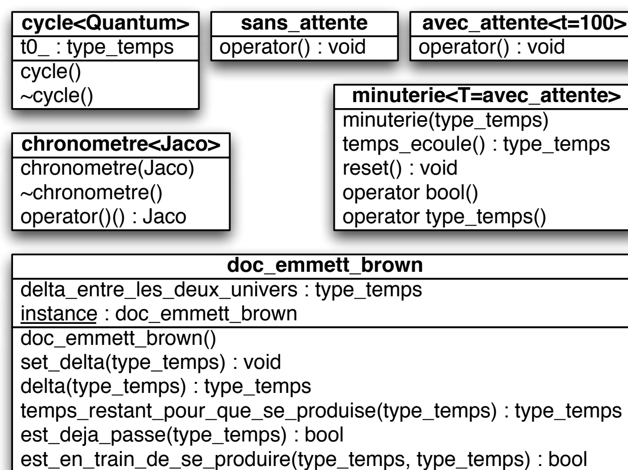

Figure 6.1 Classes gérant le temps

❶ Abstraction des opérations du temps

Nous avons créé la classe temps qui renferme le type à utiliser pour manipuler le temps (temps : :type_temps) regroupe les opérations à faire sur le temps comme obtenir le temps courant (temps : :maintenant()) ou attendre un instant (temps : :attendre(type_temps)), par exemple. La création d'une classe ne renfermant que des méthodes statiques nous a paru la solution la plus simple pour gérer le temps, qui ici est vu comme un entier. Nous aurions pu créer une classe pour représenter le temps de manière totalement indépendante de la représentation interne, mais il nous aurait alors fallu la rendre compatible avec des opérations arithmétiques et toutes sortes de choses, alors que nous sommes assurés de toujours gérer le temps comme une valeur numérique.

❷ Chronomètre d'athlétisme

Nous avons créé le patron de classe chronometre<Jaco>, qui était anciennement connu comme weather_report<Jaco>, pour représenter un chronomètre RAII appelant, à sa destruction, un foncteur de type Jaco avec en paramètre le temps écoulé depuis la création de l'objet. Nous avons fourni également les classes carbone14 (conservation du temps écoulé) et scribe (affichage du temps écoulé sur la sortie standard) comme exemples de foncteurs pouvant être utiles.

3 Minuterie de cuisine

Le programme a nécessité l'utilisation de minuteries représentées par le patron de classe minuterie<T>. Le type paramétré T représente ici une stratégie d'attente ou d'action lors de la consultation de la fin de la minuterie. Nous avons fourni deux exemples de stratégies, l'une sans_attente étant un foncteur nul, et l'autre avec_attente<t> réalisant une attente d'un certain temps en millisecondes.

4 Cycle

Nous n'avons pas à proprement parler besoin de respecter des contraintes de constance, mais nous avons tout de même créé la classe cycle<Quantum> qui sert à automatiser l'attente d'un certain temps après la réalisation d'une tâche soumise à une contrainte de constance. Un objet de type cycle est créé au début du cycle à surveiller et à sa destruction, l'objet fait attendre le fil d'exécution courant de manière à ce que le temps passé depuis l'instanciation du cycle et la destruction effective du cycle soit de Quantum.

5 Docteur Emmett Brown

Afin de résoudre les problèmes liés à l'interaction entre deux systèmes temporels distincts, nous n'avons pas pu trouver meilleur expert que le docteur Emmett Brown, du film *Retour vers le Futur*. Nous avons donc représenté le savant fou par un singleton qui pouvait être référencé par toute classe qui aurait un souci de représentation du temps et qui voudrait obtenir des informations d'une information temporelle représentée selon un référentiel distant commun.

SECTION 7

Exécution du programme

Le module d'exécution du programme est responsable de l'utilisation des autres modules de façon à ce que le programme s'exécute et possède les fonctionnalités demandées.

Il y a trois classes responsables de l'exécution de différents aspects du programme. Chacune de ces classes fonctionne sur le même principe : le constructeur exécute toutes les opérations .

1 Classe principale

mon_piano_damour est la classe qu'il convient d'instancier au démarrage du programme puisque c'est l'instance de cette classe qui a la responsabilité de l'interface graphique : la création de l'interface survient à la construction de la classe, et la fermeture

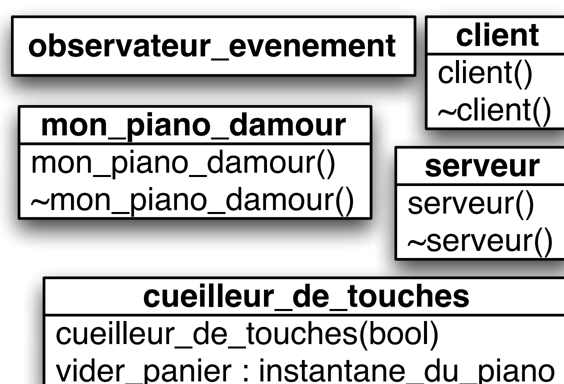

Figure 7.1 Classes chargées de l'exécution du programme

de l'interface survient à sa destruction. C'est également ici que sont définis les comportements de l'interface, le déroulement du début du programme jusqu'à l'instanciation d'un nouveau clien ou serveur.

❷ Cueilleur de touches

Le passage de la représentation en série à la représentation en parallèle de l'état du piano est géré par la classe `cueilleur_de_touches` qui reçoit des notes lorsque l'événement correspondant à l'appui sur une touche a été déclenché, que le nombre maximum de touches simultanées n'a pas été atteint et que la touche correspond à une note valide. La méthode `vider_panier()` permet de récupérer un instantané à partir des notes actuellement capturées.

❸ Client

La classe `client` renferme l'exécutif client qui démarre avec une connexion réussie au serveur, puis qui enchaîne avec la réussite du protocole de synchronisation, et enfin la mise en place d'un régime d'exécution particulier où le traitement des nouveaux événements d'interface a une haute priorité, et que périodiquement l'affichage se rafraîchit et un instantané du piano est envoyé.

❹ Serveur

La classe `serveur` renferme l'exécutif serveur qui démarre avec l'attente d'une connexion, enchaînant avec la réussite d'une connexion et du protocole de synchronisation associé. Le régime d'exécution particulier associé au serveur donne une haute priorité à l'enchaînement des tâches de récupération de nouveaux instantanés par le réseau, et de commande

de la lecture d'un son si le moment est opportun. Une priorité plus faible est donnée à l'actualisation de l'affichage et le traitement des événements de l'interface.

CHAPITRE 4

Réalisation

Ce chapitre s'attarde sur quelques aspects de la réalisation que nous n'avons pas encore détaillé ici, et qui peuvent justifier des choix que nous avons fait.

SECTION 1 Protocole de connexion

La connexion entre le client et le serveur a quelques petites particularités qu'il est bon de signaler. Tout d'abord le client et le serveur ouvrent un port UDP. Le client ouvre un port bien spécifique et connu localement par le serveur, tandis que le serveur ouvre le premier port libre que le système d'exploiation veut bien laisser ouvrir. Alors des paquets UDP bidon sont envoyés du serveur vers le client, tant qu'un temps arbitraire n'a pas été dépassé, et le client qui reçoit un de paquets reçoit également le port UDP du serveur. Les canaux de transmission sont prêts.

SECTION 2 Mécanisme de synchronisation

La nécessité de respecter la contrainte de brièveté nous a obligé à nous préoccuper de la question de l'interprétation du temps par le serveur. En effet, planifier un respect ou détecter un non-respect de cette contrainte implique de pouvoir se représenter le temps universel (à l'échelle de nos deux programmes) du point de vue du client comme du serveur. La classe `synchronisation` est responsable de la synchronisation, c'est-à-dire l'échange de donnés permettant de calculer un changement de référentiel entre le temps universel perçu par le client et celui perçu par le serveur. La formule permettant de passer du temps vu localement au temps vu par la machine distante est la suivante :

$$t_{distant} = t_{local} + \delta_t$$

Les temps $t_{distant}$ et t_{local} doivent référer au même instant vus avec un référentiel différent. En pratique, ces informations sont déterminées après une connexion réussie. C'est pour cela que l'opération de synchronisation doit être démarrée à un moment qui est le même pour les deux parties.

Il est nécessaire de connaître également le temps de latence induit par la communication réseau. Cela fait partie également du mécanisme de synchronisation, puisqu'après l'information sur δ_t obtenue, le calcul du temps de *roundtrip* (aller-retour d'un paquet sur le réseau) sera calculé. Pour le serveur, il s'agit de lancer un certain nombre de paquets (cela se compte en centaines), et à compter le nombre de paquets marqués par le client qui reviennent. Le temps d'aller retour du paquet est la différence du moment où le paquet renvoyé a été reçu et du moment inscrit au paquet lors de son envoi inital. Une moyenne sur le nombre de paquets renvoyés bien reçus est faite et cela détermine les caractéristiques de la connexion.

SECTION 3 Gestion de l'exécution des tâches

Nous avions hésité entre la mise en place de *threads* et la monoprogrammation pour mener à bien le projet. Nous avons choisi de continuer à réaliser le programme de manière monoprogrammée, car la gestion des threads liés aux ressources d'affichage SDL est calamiteuse sous Mac OS X, notre environnement de développement. Une meilleure justification, probablement, est de considérer que le programme est très statique et que ce sont toujours les mêmes tâches qui s'y exécutent.

L'enchaînement des différentes tâches dépend des priorités à respecter dans les contraintes. Le déroulement d'une action complète porteuse de sens aux yeux de l'utilisateur est décrit à la figure 3.1.

SECTION 4 Audio

Cette section s'attarde sur les réflexions que nous avons menées pour réaliser la partie audio. Nous y incluons bien entendu les réflexions antérieures à nos découvertes lors des tests que nous avons réalisé. On se place donc dans un cadre général pour les réflexions concernant le paramétrage du format des échantillons et de la carte son.

1 Préparation des échantillons audio

Les échantillons audio utilisés pour ce projet proviennent d'une banque de sons au format Soundfont[1] . Il a fallu utiliser un séquenceur et un échantillonneur afin de répartir

[1] Nous avons pris un son de kalimba, plus communément appelé *piano à pouces*, qui consiste en une petite caisse de résonance surmontée de petites lamelles métalliques qui vibrent. On en joue le plus souvent

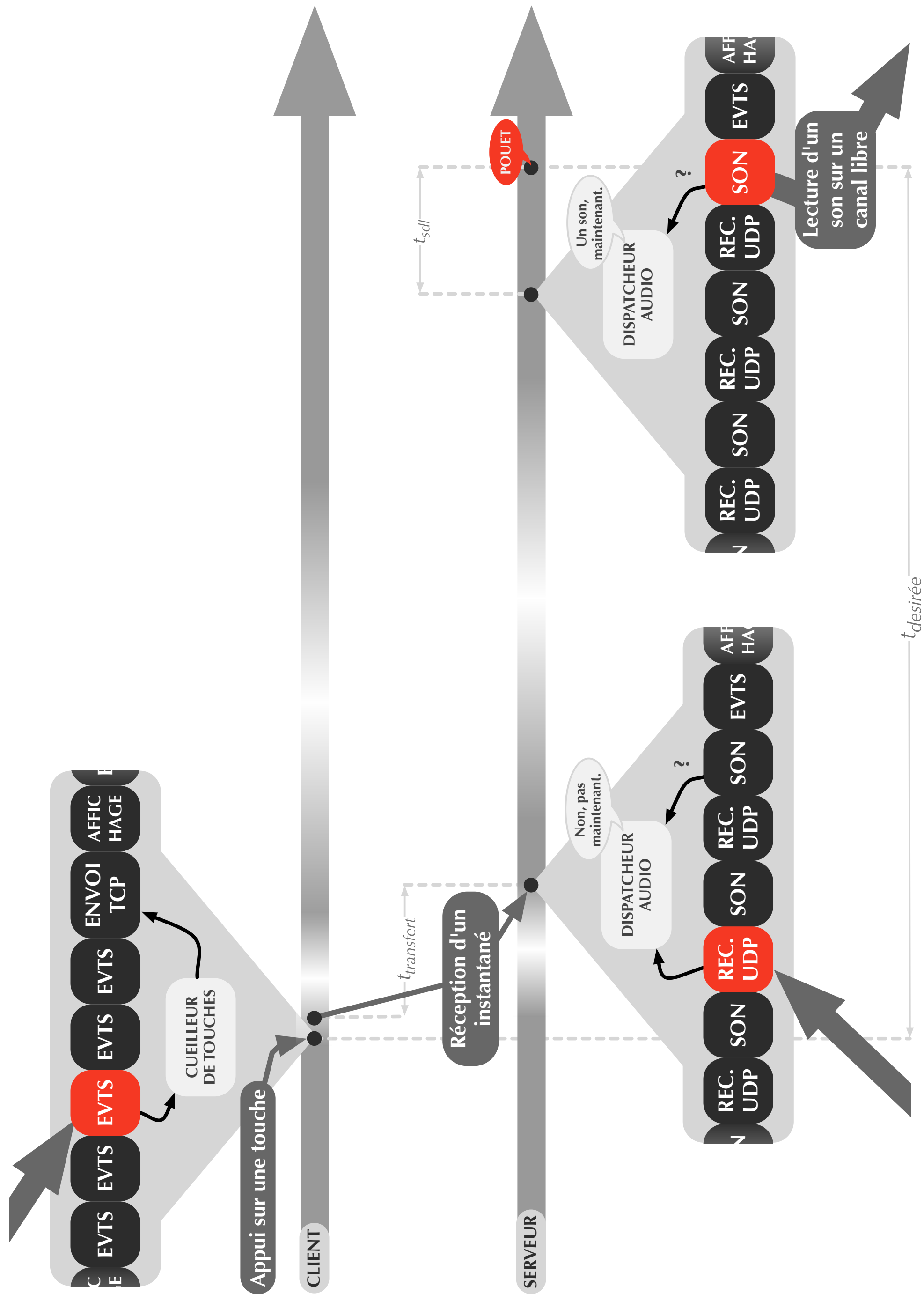

Figure 3.1 Ce que Shakespeare nommait *Da (fat) Big Picture*

la lecture de chacune des notes sur une plage temporelle assez longue pour que les notes ne se chevauchent pas. Nous avons appliqué un effet de type *noise gate* [2] sur les notes jouées afin de garantir une totale séparation entre les notes et de limiter leur résonance. Le résultat produit fut un fichier audio où toutes les notes jouées étaient présentes.

Nous avons ensuite séparé les différentes notes en utilisant un éditeur d'échantillons et leur format fut calibré sur celui de notre piano d'amour grâce à l'utilitaire en ligne de commande `sox`. Nous avions fait l'erreur d'exiger au livrable 00 que le piano d'amour ait une tessiture importante, et nous avons donc été obligés de satisfaire cette exigence en proposant une étendue royale de quatre octaves et demi (53 notes), ce qui permet de satisfaire les fantasmes musicaux refoulés de chacun sans sacrifice.

2 Chargement des échantillons en mémoire

Notre énorme banque de sons totalise 1,3 Mo de jouissance acoustique, que le piano d'amour doit charger en mémoire pendant la phase de mise en place du serveur. Le chargement est effectué grâce à la classe RAII chargeur_de_sons où chaque son est chargé sous sa forme normale et sous une forme atténuée.

3 Choix du format d'échantillonnage

Le respect de contraintes de latence dans une application audio est déjà en soi un gros problème. La latence d'un système audionumérique dépend de la fréquence d'échantillonnage et de la taille du tampon consommé par le pilote de carte son et rempli par l'application, donc elle dépend de l'application mais aussi du pilote de carte son [3]. Il est possible de choisir cette taille, dans certains cas, mais cela nécessite de prendre en compte certains facteurs liés au son.

a Taille du tampon

Un échantillon est une valeur d'amplitude du son à un instant donné. La taille d'un échantillon dépend de plusieurs facteurs déterminés par le format d'échantillonnage.

Il s'agit de la résolution, c'est-à-dire le nombre de bits attribués à chaque échantillon pour un canal, et du nombre de canaux, typiquement deux pour la stéréo, ou un seul pour le son mono.

La taille d'un échantillon est donc le produit de ces deux valeurs. La fréquence d'échantillonnage détermine le nombre d'échantillons par seconde. La taille du tampon audio est donc déterminée par un nombre d'échantillons dans un certain format, et grâce à la connaissance du format et de la fréquence d'échantillonnage actuels du pilote de la carte son on peut déduire de la taille du tampon une durée qui sera le temps de latence entre la

---

avec les deux pouces, mais bientôt tous les griots préféreront utiliser le piano d'amour pour en jouer !

[2] L'effet *noise gate* est équivalent à un ingénieur du son survitaminé qui baisse le volume lorsque le signal descend en dessous d'une certaine valeur, et qui le remonte lorsque ce signal remonte au-dessus de la valeur. Cela permet de filtrer le bruit de fond.

[3] C'est ceci précisément qui nous a posé problème pour le respect des contraintes établies au livrable 00 !

production du son dans le tampon et la lecture dans la carte son. À l'inverse, en connaissant le temps de latence qu'on souhaite ne pas dépasser, on peut déduire une taille de tampon qu'on s'engage à remplir en un temps donné[4] .

Ce serait simple comme cela, mais une taille de tampon trop petite peut occasionner des déformations du son joué, du fait que le fournisseur ne peut pas remplir le tampon dans le temps imparti, ou alors que le pilote de la carte son lui-même ne peut pas utiliser ce tampon aussi vite dans le format d'échantillonnage donné. Ce dernier cas est souvent rencontré et il est difficile de prévoir des mises au point pour chaque système et chaque pilote : les pilotes utilisés pour réaliser une latence basse et une haute qualité du son diffèrent pour chaque système d'exploitation et il n'existe pas vraiment de manière unifiée de les utiliser et de les mettre en œuvre.

La librairie Portaudio, que nous avons utilisé pour quelques tests, propose d'unifier l'accès aux serveurs audio à travers les plateformes, mais la mise à disposition d'un pilote de carte son à basse latence est essentielle pour pouvoir profiter d'une latence acceptable avec ce système. C'est notamment pourquoi nous sommes restés sur SDL_mixer, qui de plus était plus simple à utiliser.

b Format des échantillons

Ainsi, si la taille de tampon est un paramètre qu'il ne faut pas trop réduire sous peine de ne pas pouvoir jouer le son correctement avec certains systèmes, il est possible de préparer les sons dans un format de moindre qualité mais aussi moins volumineux. Cela permettrait de réduire le nombre de remplissages de tampon pour chaque son et donc d'assumer une latence correcte sans mettre en danger la lecture des échantillons. Cependant, la réduction de qualité des sons, qui au passage ne signifie pas *les convertir en MP3* mais changer les caractéristiques du format, les données restant non compressées, ne doit pas rendre les sons inutilisables. Il faut donc paramétrer cette réduction de qualité de manière raisonnée.

Diviser par deux le nombre de canaux revient à transformer le son stereo en son mono, et pour notre piano d'amour c'est acceptable puisque nous ne faisons pas un *DX7 d'amour*[5] . Cela revient également à diviser la taille des sons par deux, ce qui est un gain substantiel. Réduire la résolution revient à accorder moins de place pour coder chaque échantillon. Cela a pour effet de réduire le nombre de valeurs que peut prendre chaque échantillon, ce qui se traduit à l'écoute par une réduction non pas de la plage dynamique mais de sa granularité : les sons sont moins bien définis. Le cas extrême d'une telle réduction donne des sons de console Game Boy[6] .

Enfin, réduire la fréquence d'échantillonnage revient à décrire chaque seconde de son avec moins d'échantillons. Cela entraîne la perte d'informations fréquentielles, ce qui per-

[4] Ce contrat est une forme de respect de contrainte de constance : le tampon doit être rempli à chaque intervalle. Un non-respect de la contrainte entraînerait la lecture de n'importe quoi par le pilote de carte son, donc potentiellement des craquements, des *bzit*, des *ni !*, des *pouet* et des *ecky-ecky-ecky-pakang-zoom-ping*.

[5] Le Yamaha DX-7 mkII est un synthétiseur stereo à modulation de fréquence qui a été très utilisé par les musiciens dans les années 80.

[6] Cela peut être un effet recherché, qui est utilisé par exemple dans les styles de musique électronique *breakcore* ou *chiptune*.

met de distinguer si le son a des composantes graves ou aiguës. Le son représentera moins de fréquences et cela se traduirait à l'écoute par un son plus sourd et moins bien défini. Il est à noter que le théorème d'échantillonnage de Nyquist-Shannon montre de plus que pour que le son soit composé d'une fréquence f, il doit être échantillonné au minimum à $2f$. L'audition humaine reconnaît les sons qui ont des fréquences entre 16 Hz et 20 kHz. Pour que le son échantillonné soit équivalent au son réel pour une oreille humaine, il convient donc de l'échantillonner à environ 40 kHz, ce qui est inférieur à la fréquence d'échantillonnage d'un CD audio[7] . Il est à noter que pour une même taille de tampon, la latence diminue quand la fréquence augmente. Cependant, cela signifie que pour le même nombre d'échantillons, le tampon devra être vidé plus de fois si la fréquence d'échantillonnage est plus grande.

Afin de faire un compromis entre la qualité du son et l'occupation des ressources, nous avions déterminé empiriquement que des sons mono à 22 kHz et 16 bits de résolution étaient acceptables pour notre piano d'amour.

[7] Pour un CD audio, le son est échantillonné sur deux canaux, à 44,1 kHz et avec une résolution de 16 bits.

CHAPITRE 5

Tests et calibration

Ce chapitre concerne les tests que nous avons mené au sujet des différents axes de notre piano d'amour. Nous les présenterons de manière à identifier la méthodologie de tests, les objectifs des tests et les décisions qui en découlent, ces dernières permettant de calibrer le système afin qu'il puisse garantir le respect des contraintes temps-réel.

Partie audio

1 Motivations

Il est une chose de pouvoir demander à la carte son, *via* une librairie, un ensemble de caractéristiques que présentera le flux audio nouvellement ouvert pour notre programme, mais c'en est une autre d'obtenir les caractéristiques demandées. La décision du pilote de carte son, transmise à la librairie puis éventuellement à notre programme, dépendamment de la librairie utilisée, est sans appel, et cela dépend non seulement des capacités de la carte son mais aussi du pilote lui-même.

Les librairies génériques pour le son comme SDL_mixer ou Portaudio sont généralement avares en renseignements sur les caractéristiques du flux dont l'ouverture a été demandée, et les informations à tirer sont minces en raison de la variété des informations disponibles sur une plateforme ou une autre. Il n'était donc pas sage de se fier à ces caractéristiques obtenues dans le sens où certaines données auraient été erronées pour certaines plate-formes. Ce qui est embêtant avec cela est qu'il est par exemple impossible d'obtenir la taille de tampon réellement allouée de manière multiplateforme, à moins de récrire une librairie ou d'accepter d'abandonner certaines plateformes ou certains types de pilotes de carte son.

N'ayant pas vraiment de temps pour cela, nous avons préféré tester la latence réelle obtenue en fonction de divers paramètres demandés, dans des conditions normales de température et de pression, afin de pouvoir donner une taille de tampon acceptable et réaliste pour un échantillon que nous considérerons représentatif de l'ensemble des pla-

teformes cibles[1] .

Les tests permettront de donner des chiffres en face des évaluations subjectives, et même de déterminer une condition nécessaire pour le respect des contraintes temps réel.

2 Méthodologie de test

Le test consiste à lancer un sous-programme dont on peut trouver les restes dans `tests/-latence_audio.cpp`, qui joue un son particulier sur les haut-parleurs lorsqu'un caractère a été accepté en provenance du flux d'entrée standard.

Pendant ce temps, un autre programme[2] , sur ordinateur local ou sur un autre à proximité selon la configuration disponible, enregistre ce qui provient du microphone intégré. Ce dernier capte le son en provenance des haut-parleurs incriminés et du clavier dont il est indispensable de pouvoir entendre les cliquetis.

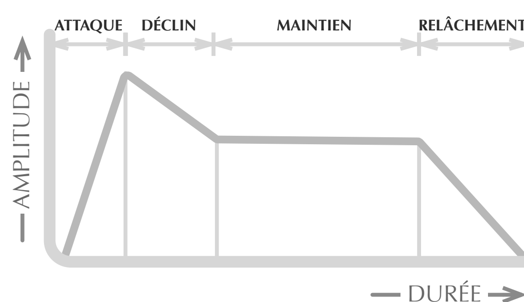

Figure 1.1 Quatre moments bien particuliers dans la vie d'un son

Un échantillon de test s'obtient en pressant doucement la touche d'un caractère, disons D, puis en activant vigoureusement la touche Entrée de manière à ce que le son percussif de l'activation de la touche soit visible nettement sur la représentation graphique du son enregistré. La forme d'onde du son qui aura été joué sera reconnaissable également sur l'enregistrement. Il faudra mesurer le délai entre le son produit par l'appui de la touche et le son joué, mais il faudra tenir compte des caractéristiques des sons. Grossièrement, on peut définir quatre moments dans la vie d'un son percussif, soit l'attaque, le déclin, le maintien et le relâchement[3] , qui sont représentés à la figure 1.1. Le son de kalimba utilisé est un son percussif avec une attaque bien déterminée mais qui change selon la note choisie. Nous nous arrêterons sur la note **Do 3** qui a une attaque de 25ms.

3 Résultats des tests

Nous avons effectué deux séries de tests sur les machines Macbook Pro et Lenovo :

---

[1] Bien entendu, avec trois ordinateurs, ce n'est pas très représentatif, mais les cartes son de tous les ordinateurs sont en général de la même piètre qualité (*argumentum ad Google*) et offrent toutes des garanties à la fois similaires et inexistantes...

[2] Pour nos tests, nous avons utilisé le logiciel libre Audacity disponible sur une multitude de plateformes.

[3] Les synthés et les anglophones, de manière générale, parlent plutôt d'**ADSR** : attack, decay, sustain,

Taille de tampon	t_{sdl}	$t_{portaudio}$	Évaluation subjective	CN TR
2048 échantillons	145ms	120ms	Inconfortable	Non
1024 échantillons	75ms	60ms	Presque inconfortable	Non
512 échantillons	55ms	45ms	Moyennement confortable	Non
256 échantillons	55ms	42ms	Moyennement confortable	Non

Figure 1.2 Résumé des tests de la partie audio

- lecture d'une note en utilisant SDL_mixer ;
- lecture d'une note en utilisant Portaudio.

Les canaux de sortie par défaut ont été utilisés à chaque fois pour être plus représentatif de l'utilisation du système en production. Malgré la différence entre les systèmes de son de Mac OS X et de Ubuntu, il est apparu que les résultats ont été plus ou moins identiques. La figure 1.2 reprend les valeurs moyennes des temps calculés en fonction des caractéristiques demandées.

Il est ressorti des résultats qu'il était inutile de demander une taille de tampon inférieure à 512 échantillons, et certains conseillent 1024 échantillons comme limite inférieure pour être certain d'obtenir ce qu'on demande sur tous les systèmes. 1024 échantillons constituent donc une sécurité pour que la réponse du piano d'amour soit uniforme à travers les plateformes.

4 Prises de décision

Portaudio donne globalement de meilleurs résultats mais ce n'était pas encore assez bon pour respecter la première version des contraintes temps-réel. Dans tous les cas, il était donc impossible de tenir les promesses faites dans la première spécification des contraintes. Nous avons donc trouvé d'autres contraintes qui étaient pour nous compatibles avec les circonstances techniques, la facilité d'installation et la réflexion sur les systèmes temps réel. Nous avons choisi de privilégier SDL_Mixer à Portaudio en raison de sa plus grande facilité d'utilisation, même au prix d'une charge supplémentaire qui somme toute n'a que peu d'incidence sur le respect de nos nouvelles contraintes.

SECTION 2 Partie réseau

1 Motivations

Notre livrable 00 stipulait que le piano d'amour devait fonctionner en respectant les contraintes temps réel à condition de disposer d'un lien réseau correct, sans préciser la

release.

Situation	$t_{roundtrip}$	$t_{transfert}$	Pertes	Variabilité	CN TR
localhost	2ms	1ms	0%	Faible	Oui
Réseau public 802.11	500ms	250ms	90%	Élevée	Non
Réseau ad-hoc 802.11	200ms	100ms	10%	Moyenne	Non
Ethernet 10Mbps	3ms	1,5ms	0%	Faible	Oui

Figure 2.1 Résumé des tests de la partie réseau

nature de ce qui est correct. Afin de vérifier si le piano d'amour respecte les contraintes temps réel dans une situation correcte, il nous a fallu déterminer quelles circonstances en termes d'infrastructure réseau constituent une situation correcte.

2 Méthodologie de test

Nous avons sélectionné plusieurs possibilités de mise en réseau de deux ordinateurs, qui sont considérées intuitivement comme plus ou moins fiables ou rapides. Grâce au sous-programme dont les restes se trouvent dans `tests/calcul_roundtrip.cpp` et qui utilise le protocole de synchronisation du piano d'amour, nous avons lancé les programmes client et serveur sur deux ordinateurs selon les configurations suivantes :

localhost Un ordinateur qui lance les programmes client et serveur avec l'adresse `localhost`.

Réseau public 802.11 Deux ordinateurs connectés au même point d'accès sans fil partagé dans tout un immeuble et protégé par une clé WEP.

Réseau ad-hoc 802.11 Deux ordinateurs connectés au même réseau ad-hoc sans sécurité.

Ethernet 10Mbps Deux ordinateurs connectés directement avec un câble Ethernet 10Mbps.

Firewire 800 Nous n'avons pas eu le temps de tester cette possibilité, qui doit être très peu utilisée et qui n'a que peu d'intérêt étant donnée la longueur d'un cable Firewire 800.

3 Résultats des tests

La figure 2.1 reprend les moyennes des résultats pour chaque situation, à savoir le temps de *roundtrip* (aller-retour d'un paquet), le taux de pertes de paquets et la variabilité du temps de transfert.

4 Prises de décision

Il est ressorti des tests que l'utilisateur du piano d'amour qui utilise autre chose qu'une connexion filaire ne peut pas compter sur les garanties de respect des contraintes TR, dans

le cas où une latence plutôt basse a été demandée. Cependant, les garanties restent valides tant que la latence demandée est assez haute.

SECTION 3

Mise en situation réelle

Nous avons réalisé des tests de mise en situation très similaires au test de la partie audio afin de vérifier que le piano d'amour remplissait bien les contraintes temps-réel. Nos mesures confirment le bon respect des contraintes avec une marge d'erreur de 10ms. Sur la base de ces observations, nous considérerons notre système comme un système temps-réel en guimauve.

CHAPITRE 6

Conclusion

Définir un projet de système en temps réel dans l'espace d'un trimestre est difficile mais le réaliser l'est encore davantage. Nous avons dû nous y prendre à plusieurs fois pour spécifier les contraintes, définir une manière de faire interagir les différents modules, donner une place à chacun d'eux dans le temps, car nous étions face à notre première expérience dans ce domaine. Nous avons donc refait plusieurs choses et nous en avons raffiné d'autres, nous avons même beaucoup tourné en rond, ce qui a pris beaucoup de temps, car nous faisions de grands ronds. Une fois que nous avons su quels éléments étaient pertinents et lesquels étaient secondaires, nous avons constaté une faiblesse dans nos choix qui nous obligeait soit à tout revoir, soit à changer les contraintes. Nous pensons que le fait d'avoir mêlé des problèmes de réseau à des problèmes de son ne nous a pas facilité la tâche de compréhension et d'analyse, et de plus ce type de contrainte est assez différent de ce à quoi nous nous attendions en appréhendant les systèmes temps réel, où l'on pense plus à la constance ou à la régularité.

Nous avons pu également mettre en pratique ce que nous avons appris sur le langage C++ avec lequel nous avons été réconciliés grâce à ce cours. Nous avons essayé de peser le pour et le contre de chaque technique et possibilité à notre disposition, sans pour autant y arriver à chaque fois. Nous sentons que le projet nous a grandement servi pour progresser autant dans le domaine purement technique qu'au niveau conceptuel.

Pour finir, voici les partitions de trois morceaux de musique que nous aimons jouer sur notre instrument et que nous vous invitons à pratiquer en QWERTY.

Son des îles Répéter alternativement A et D très vite, puis S et F de la même façon, puis D et G et ainsi de suite en avançant et reculant. Bienvenue sur une île paradisiaque !

Quatre saisons WT UU YT O (silence) JU UU YT O (silence) JU JOJ UY W (silence) (bis)

Jazz-funk En accords : (FHK ;)×2 (DGJL) (silence) (SFHK)×2 (DGJL) (silence) (FHK ;)×2 (DGJL) (silence) (SFHK)×2 (ADGJ)

Ce document a été produit à l'aide des logiciels libres LyX et XƎTEX.