UNIVERSITE LIBRE DE BRUXELLES

FACULTE DES SCIENCES

DEPARTEMENT D'INFORMATIQUE

MIDI Sans Frontières

Analyse des problèmes relatifs à la communication musicale sur IP

Mémoire présenté en vue de
l'obtention du grade de
Licencié en Informatique

Directeur : Raymond Devillers
Présenté par : André Van Schel

Année Académique
2001 - 2002

Abstract

Nous proposons, dans le cadre de ce mémoire, d'analyser la possibilité d'intégrer le support d'un langage de description du jeu musical sur un réseau de type IP. Nous commencerons par montrer l'intérêt d'une telle approche à travers différents modèles d'utilisation.
Après une étude de la syntaxe du langage MIDI que nous avons choisi d'adapter pour permettre la communication interactive, nous nous intéresserons aux aspects de la perception qui rentrent en jeu dans la détection des imperfections de transmission.

Par une analyse systématique de scénarios illustrant l'apparition de problèmes de transmission, nous déterminerons les mécanismes à mettre en oeuvre afin de pouvoir minimiser leur incidence.

Nous étudierons ensuite le choix d'une architecture réseau apte à supporter la transmission MIDI, ce qui nous amènera à faire un parallèle avec les implémentations de la téléphonie sur IP pour ensuite nous demander si l'intégration de nos développements dans le cadre du protocole H.323 ne constitue pas un atout indéniable.

L'étude de la gestion de la qualité de service offerte par les réseaux en fonction de notre type d'application nous a finalement paru un détour essentiel pour passer à la phase d'implémentation qui nous permettra de juger de l'applicabilité des concepts abordés.

Remerciements

Je tiens tout d'abord à remercier, M. Vassilis Constantopoulos et M. Joël Cannau, pour la confiance, l'aide et les encouragements qu'ils m'ont accordés durant l'élaboration et la rédaction de ce mémoire.

Je tiens également à remercier mon directeur de mémoire, M. Raymond Devillers.

Je remercie, Mme Marie-Ange Remiche, M. Claude Machgeels et M. Joël Goossens pour avoir accepté de faire partie du jury ; Ainsi que Mme Carolyn Drake et M. Bruno Repp pour l'aide qu'ils m'ont apportée dans le domaine de la perception auditive.

Je voudrais encore remercier Isabelle, Thomas, Christophe, Toan et plus particulièrement Laura qui n'ont pas manqué à cette occasion de me prouver, une fois de plus, leur amitié.

Je remercie enfin chaleureusement ma famille, pour son éternel support.

Merci !

Table des matières

1 Introduction 8

2 Le but de l'interaction musicale 11

2.1 Le modèle Internet 11

2.2 Le modèle Studio 12

2.3 Le modèle Serveur de sons 13

3 Le protocole MIDI 15

3.1 Comparaison audio et MIDI 15

3.2 La communication MIDI 17

3.3 Les messages MIDI 19

Les channel messages 20

Les system messages 21

4 Perception musicale 24

5 Analyse des problèmes de transport MIDI... 28

5.1 Les retards 29

Cas 1 : Retard constant 29

Cas 2 : Retard variable 30

Cas 3 : Inversion 31

5.2 Les pertes 32

Cas 1 : Perte du Note On 33

Cas 2 : Perte du Note Off 34

Cas 3 : Perte du Note On et du Note Off correspondant 35

5.3 Cas de perte et de retard plus ambigus 35

Cas 1 : Inversion 36

Cas 2 : Perte du Note On et du Note Off correspondant 36

5.4 Transport d'autres types de message 37

5.5 Conclusion de l'analyse des problèmes de transport 38

6 Intervalle de transmission 40

7 Pré-Analyse du comportement MIDI 44

8 Vérification de l’état du système.................. 49

9 Mécanisme d’anticipation............................ 51

10 Choix de l’architecture réseaux................... 55

10.1 Objectifs et contraintes 55

10.2 Yamaha mLAN 56

10.3 Choix du modèle TCP/IP 59

La couche Internet 60

La couche transport 62

11 RTP / RTCP.. 66

11.1 RTP 66

La session RTP 67

Le paquet RTP...67

Les fonctionnalités de RTP 69

11.2 RTCP Real-Time Control Protocol......................................70

12 MIDI Versus VoIP 73

13 H.323... 75

14 QoS 77

14.1 Gestion de la QoS 79

Gestion en meilleur effort..79

Gestion du trafic au sein du réseau 80

Gestion du trafic aux extrémités du réseau...81

14.2 Niveaux de service de la QoS réseau...................................81

14.3 Les modèles liés au protocole IP...83

IntServ (RSVP) 83

Diffserv 84

15 Mesure de la qualité.................................... 85

16 Implémentation 88

16.1 Mise en place des éléments de base...................................88

Acquisition et reproduction de données MIDI...89

Simulation..90

Expérimentation..93

Générateur de notes 94

Emetteur Récepteur RTP..94

Le relais UDP 95
16.2 Mise en place de mécanismes de correction 97
Correction basique 98
Transmission de l'état de l'émetteur 99
Intervalle de transmission de l'état du système 100
Temporisation 101

17 Conclusion 104

Bibliographie 105

Annexes 109
I Code source de l'application 110
II Index des figures et des tables 148
III Liste des acronymes utilisés 150
IV Glossaire 152

1 Introduction

Les plus grands changements qui s'opèrent actuellement dans nos sociétés occidentales se caractérisent bien souvent en terme d'information et de communication ainsi qu'en une mondialisation croissante des échanges.
Il peut être étonnant de voir à quel point ces changements concernent tout à la fois les mentalités, les sciences et la technique. Ces différents aspects semblent évoluer dans un même sens.

Parmi ces changements, un des plus spectaculaires et sans nul doute à l'origine de bien d'autres transformations concerne l'informatique. Après s'être répandue dans nos entreprises et chez les particuliers pendant les années quatre-vingts, l'informatique a elle-même été sujette à une intense transformation au cours de la dernière décennie pour se matérialiser aujourd'hui comme un puissant vecteur de communication et d'échange. Avec le développement du réseau Internet, elle change petit à petit notre façon d'apprendre, de comprendre et de connaître. Elle transforme de façon importante certains processus sociaux et a ainsi fait rentrer chaque jour de nouveaux mots dans le langage courant, tels que télétravail, vidéo-conférence, home-shopping, télé-apprentissage, paiement électronique ou bibliothèque électronique.

Par ce biais les frontières géographiques, sociales et même parfois culturelles s'amenuisent de jour en jour, parfois dangereusement, tant l'impact social de ces " avancées " semble être omis.

Néanmoins, il est bien compréhensible que le développement de

l'informatique dans les domaines du multimédia et des réseaux, à l'origine de bien d'autres changements, suscite un grand enthousiasme.
Les réseaux informatiques se retrouvent eux-mêmes au cœur d'une inévitable convergence, pour tendre vers un support de tous les médias par une même infrastructure. Ceci pour des raisons économiques (économies d'échelle en particulier), de simplification, de gestion et d'efficacité d'exploitation. Ainsi, de nombreux efforts sont donc aujourd'hui consentis pour la mise en place de réseau supportant aussi bien le transfert de la voix et de l'image que des données. On trouvera donc une large gamme de technologies qui approche au mieux ces buts: ATM, VoIP, H.323, SIP, ISDN, UMTS, etc.

Si la musique est probablement le mode de communication le plus universel, il est normal qu'à ce titre elle occupe une place importante au sein de ce que l'on nomme le multimédia. Internet est sans nul doute devenu la bibliothèque la plus imposante et donne ainsi accès à une somme astronomique d'œuvres musicales en tout genre.
La révolution technologique dans le domaine de la musique est sans répit. Les supports de données, les formats de fichiers, les protocoles de transfert, et les instruments de musique sont sans cesse réétudiés afin de permettre de nouvelles fonctionnalités.

Si l'auditeur ne peut que se féliciter de cette technologie lui permettant d'accéder de tout lieu et le plus rapidement au contenu musical (MP3, *streaming*, MIDI, etc.), il semble cependant que les musiciens amateurs ou professionnels n'aient pas encore l'opportunité de dépasser les frontières physiques de leur local de répétition ou de leur studio lorsqu'ils désirent composer ou être accompagnés.
Il semble pourtant n'y avoir qu'un pas pour permettre à ceux-ci de

réellement tirer parti de toute cette mouvance technologique.

L’objectif de ce mémoire est d’analyser la possibilité d’offrir au musicien un moyen d’interaction efficace à travers les réseaux.

2 Le but de l'interaction musicale

Nombre de technologies sont efficacement exploitées lorsqu'il s'agit de transmettre statiquement l'information sonore ou musicale. La transmission sous cette forme intervient après le travail de l'artiste, lorsque les différentes phases de la composition et/ou de l'interprétation ont abouti au résultat prêt à être « consommé » par l'auditeur.

Nous situons notre étude en amont, lorsque le travail de composition ou d'interprétation musicale est en cours. Nous souhaitons offrir aux musiciens une capacité d'interaction musicale, entre eux ou avec leur matériel, localement ou à distance, pour peu qu'ils disposent d'un accès à un réseau informatique permettant d'échanger les informations utiles.

Tentons d'établir l'intérêt de la démarche à travers un éventail non-exhaustif d'utilisations de cette capacité d'interaction.

2.1 Le modèle Internet

La capacité d'interaction doit permettre aux musiciens physiquement distants de jouer ensemble et simultanément sur leur instrument respectif. C'est ce qui est appelé dans certaines études le télé orchestra [A1]. Avec un tel support, on peut imaginer qu'un claviériste londonien peut se connecter par l'intermédiaire d'un réseau informatique hétérogène à un percussionniste africain afin de communiquer, musicalement. Evidemment l'efficacité d'une telle

communication dépendra fortement des performances, de la charge des lignes et des nœuds rencontrés sur le réseau traversé. On peut également intégrer dans ce modèle l'utilisation d'Internet pour le télé-apprentissage de la musique comme le suggère l'approche de J.P. Young «Piano Master Classes via the Internet». [A2]

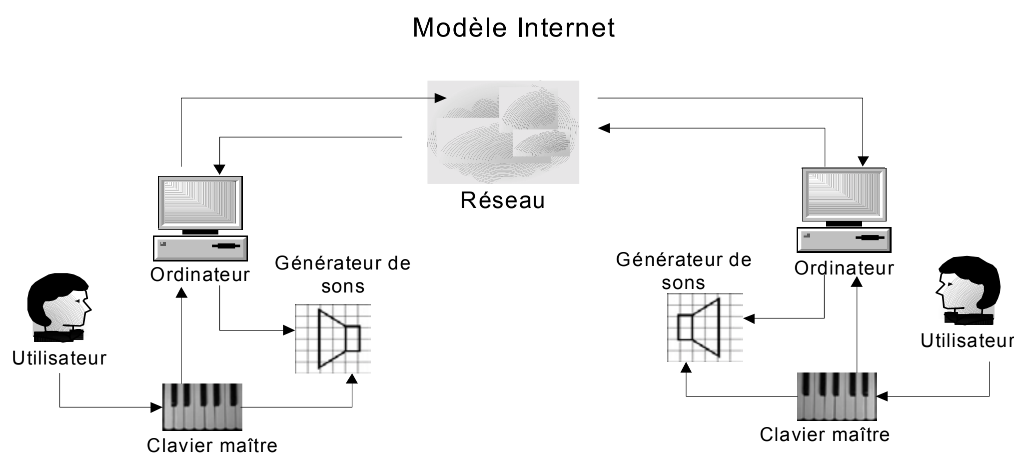


Figure 1 : Le modèle Internet

2.2 Le modèle Studio

La capacité d'interaction, prévue pour être supportée sur un réseau, permet également la simplification des multiples connexions établies dans les studios musicaux. En effet, à l'heure actuelle la diversité des câblages et des types de données échangées (données, audio numérique, audio analogique, MIDI …) ne font que rendre beaucoup plus complexe la gestion de ce type d'infrastructure. L'idée est dans ce cas d'adjoindre à ces équipements musicaux toutes les fonctionnalités pour supporter un protocole réseau classique au même titre qu'une imprimante réseau et donc de ne plus trouver dans le studio qu'un « simple réseau » informatique supportant tous les échanges. Nous montrerons également comment les schémas

d'adressage des réseaux informatiques permettent de s'affranchir de la limite du nombre d'instruments accessibles d'un point à l'autre dans les architectures de réseaux MIDI actuels.

On peut noter l'approche en ce sens du constructeur Yamaha dans le développement de son système mLAN (Voir 10.2) permettant de supporter le MIDI et l'audio sur une infrastructure basée sur le protocole IEEE1394 (FireWire).

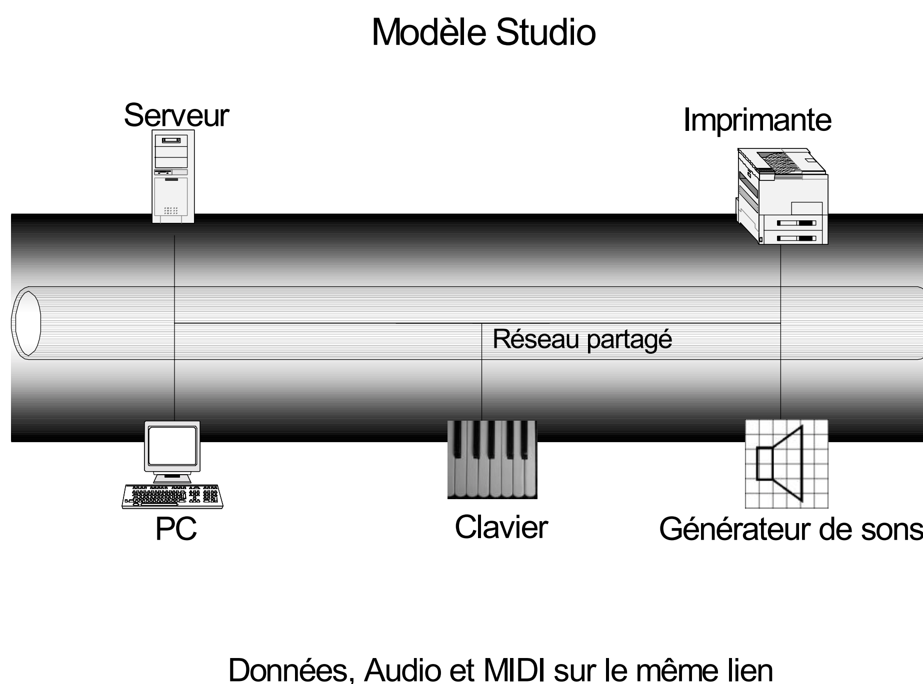


Figure 2 : Le modèle Studio

2.3 Le modèle Serveur de sons

Nous pouvons envisager la mise en œuvre sur des réseaux suffisamment rapides d'un système permettant d'avoir de très

puissants synthétiseurs sonores qui à la réception d'informations en provenance de musicien transforme celles-ci en signaux audio et les retransmette vers leur expéditeur. Ceci consiste en quelques sortes à avoir une sorte de terminal musical "bon marché" et de partager les ressources sonores d'un serveur. Il faut dans ce modèle un réseau rapide car les informations retournées souffrent d'un besoin de qualité qui implique un haut débit ainsi qu'un besoin de temps réel.

Modèle Serveur de sons

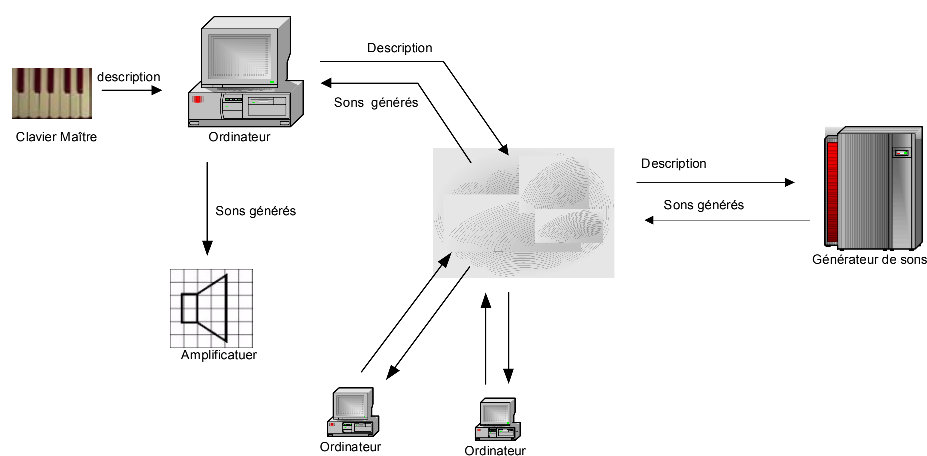


Figure 3 : Le modèle serveur de sons

3 Le protocole MIDI

La norme MIDI [I1] (*Musical Instrument Digital Interface*) définit une interface et un protocole permettant de mettre en relation différents instruments de musique afin qu'ils puissent communiquer. MIDI est largement accepté et utilisé par les compositeurs et musiciens depuis sa conception en 1982/1983.
Les instruments de musique dotés de l'interface MIDI ont donc la possibilité d'échanger des messages qui ne contiennent pas le son tel qu'il doit être produit, mais une indication sur l'action à effectuer.
MIDI a également évolué pour devenir une méthode de représentation électronique archivable du jeu musical qui rend ce protocole aussi intéressant pour les compositeurs et musiciens que pour l'utilisation dans les applications informatiques qui produisent du son.

3.1 Comparaison audio et MIDI

Contrairement aux systèmes audio, MIDI se contente d'indiquer aux générateurs de sons avec lesquels il travaille (localement ou à distance) les opérations qu'ils doivent effectuer.
MIDI fournit aux générateurs de sons des informations équivalentes à celles que l'on pourrait relever sur une partition. C'est la différence qui existe entre le fait de jouer d'après une partition sur un piano et l'action de placer un CD audio dans un système Hi-Fi afin d'en découvrir le contenu.

Alors qu'en audio numérique, Shannon nous indique qu'un son

(notion plus générale qu'une note) doit être échantillonné et mesuré à une fréquence double de sa fréquence maximale pour pouvoir le restituer fidèlement, une note en MIDI ne sera représentée que par quelques valeurs numériques : sa hauteur en fréquence matérialisée par un numéro de touche sur un clavier, l'intensité à laquelle elle doit être générée, ses instants d'apparition et de disparition, ainsi que le numéro de forme d'onde que le générateur doit utiliser. (Bien que cette dernière information ne soit envoyée qu'une fois pour toute la durée de l'utilisation de ce timbre). Ces informations suffiront au générateur de son pour faire le reste.

Il n'est donc pas difficile de comprendre pourquoi la représentation MIDI a une taille beaucoup plus compacte que l'information musicale équivalente dans le format audio le plus compressé.

Il est cependant essentiel de remarquer que le faible volume occupé par les donnés MIDI se traduit par l'absence d'informations relatives à la nature même du son utilisé (la spécificité du timbre). La production sonore réelle, orchestrée par MIDI, sera donc fortement influencée par les caractéristiques du générateur de sons utilisé.
Cette différence de qualité met en évidence un phénomène assez rare : en transmettant un message, la qualité de restitution du son transmis par MIDI peut être plus élevée à la réception si l'instrument qui envoie les instructions est de moindre qualité.

Afin de permettre l'échange de fichier standard MIDI et d'éviter de rendre les différences d'interprétation d'un générateur à l'autre trop contraignantes, la norme Général MIDI définit un ensemble de 128 timbres répondant à certaines caractéristiques bien précises que les générateurs MIDI conformes doivent implémenter. Cependant la qualité de synthèse de ces sons pourra varier d'un instrument à l'autre et une gamme plus spécifique à chaque constructeur pourra se

retrouver à côté de cet ensemble de sons.

	MIDI	DIGITIZED AUDIO
Volume	+ Faible	- Comparativement élevé
Reproduction sonore	- Dépend fortement du générateur	+ Relativement fidèle. (DAC et amplification peuvent générer des différences)
Modification	+ Edition très simple	- Traitements complexe du signal

Table 1 : Comparaison MIDI et Audio

3.2 La communication MIDI

MIDI fournit un moyen standard et efficace pour transporter les données musicales sous forme électronique. Les informations MIDI sont transmises dans des « Messages MIDI ». L'équipement qui reçoit le message doit générer le son réel.

La spécification MIDI 1.0 [I1] fournit une description complète du protocole MIDI.

Le flux de données MIDI est un flux de bit asynchrone unidirectionnel à 31.25 Kbits/sec, avec 10 bits transmis par mot (un bit Start, 8 bit de données, et un bit Stop).

La communication MIDI se déroule entre équipements qui peuvent jouer les rôles de maîtres et/ou d'esclaves. L'équipement maître doit être capable d'émettre des informations vers l'esclave qui pour sa part doit être capable de pouvoir les traiter. Pour être maître, l'équipement doit posséder une sortie (MIDI OUT), pour être esclave l'équipement doit posséder une entrée (MIDI IN). Les messages sont envoyés de la sortie (MIDI OUT) du maître vers l'entrée (MIDI IN) de l'esclave au travers d'un lien unidirectionnel, matérialisé par un câble coaxial. On notera qu'une communication bidirectionnelle implique la présence de deux liaisons physiquement distinctes entre les équipements.

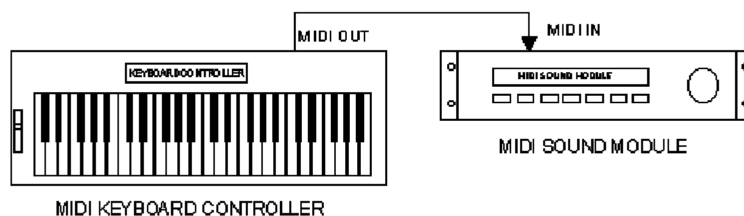


Figure 4 : Configuration simple, Maître - Esclave

Afin de pouvoir connecter en chaîne plus de deux équipements, on trouve souvent à côté de l'entrée MIDI IN, une sortie de relais (MIDI THRU) qui retransmet l'information telle que l'équipement l'a reçue à l'équipement suivant dans la chaîne.

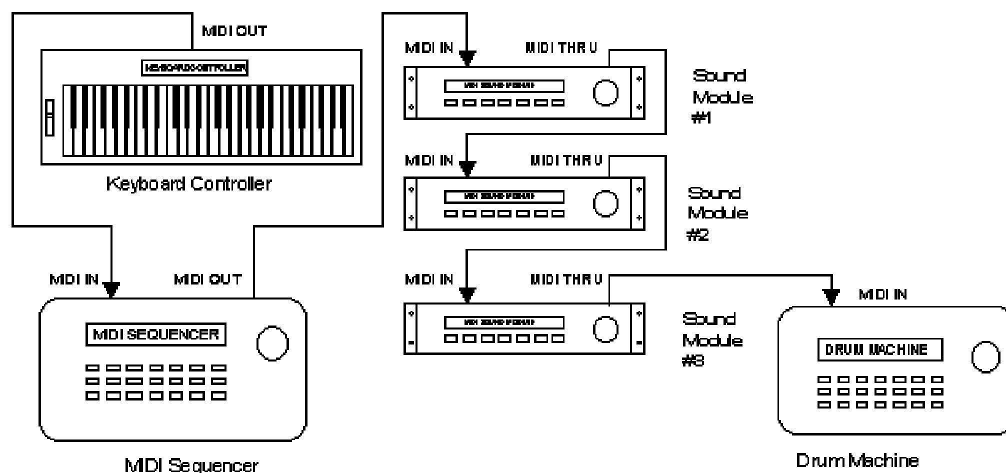


Figure 5 : Chaînage MIDI

Un instrument MIDI classique fonctionne sur 16 canaux parallèles. On peut affecter un timbre différent à chaque canal et donc le générateur peut produire un ensemble de sons variés simultanément. Cette propriété est appelée la multi-timbralité.

Etant donné l'aspect série de la communication, on peut mettre en évidence l'aspect multiplexage qui s'applique dans la mesure où 16 canaux différents utilisent le même lien. Il s'agit donc d'un multiplexage temporel statistique étant donné que les infos ne sont envoyées qu'au moment ou une émission est nécessaire.

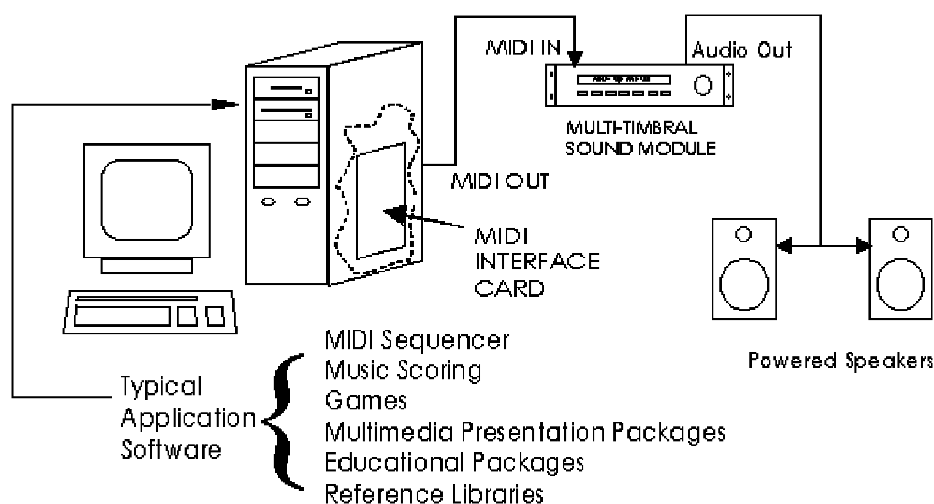


Figure 6 : Configuration MIDI avec ordinateur

3.3 Les messages MIDI

Les messages MIDI [A3] se composent d'un byte de statut (*status byte*) qui est généralement suivi par un ou deux bytes de données (*data bytes*).

On trouve deux classes principales de messages MIDI : ceux qui concernent un canal spécifique, *channel messages* (canal indiqué dans le byte de statut), et les *system messages* plus généraux.

Les channel messages

Les channel messages sont de deux types : channel voice et channel mode.

Channel voice

Les *channel voice messages* transportent l'information musicale qui doit être délivrée au générateur de son.

- Le *Note On* signale l'activation d'une touche (numéro de canal dans le byte de statut, le numéro de la touche enfoncée et la vélocité (pression exercée) dans chacun des deux bytes suivants).

- Le *Note Off* est transmis lorsque la touche précédemment activée est relâchée. Il indique également le numéro de touche, ainsi que la vélocité avec laquelle la touche avait été actionnée (généralement ce paramètre est ignoré par les instruments).

- L'*Aftertouch* : sur certains claviers MIDI, il est également possible de mesurer la pression exercée lors du relâchement des touches. (contrôle du vibrato). Selon le type de clavier on utilisera un message :

- Le *Polyphonic Key pressure*: les informations de pression sont différentes pour chaque touche : *channel* + *key* + *pressure*.

- Le *Channel Aftertouch* : une info de pression unique est utilisée pour tout le clavier : *channel* + *pressure*.

- Le *Pitch Bend* indique les changements de position de la roulette d'un clavier qui permet d'offrir un passage plus progressif d'une note à l'autre. Deux bytes de données offrent une précision suffisante pour que le déplacement de la « roulette » paraisse continu.

- Le *Program Change* spécifie le « type d'instrument » qui doit être utilisé par le générateur pour produire les sons d'un canal donné. Ce message ne nécessite qu'un byte de données pour indiquer le nouveau numéro de programme : 128 instruments possibles par banque de programme.

- Le *Control Change* permet de contrôler une large variété de fonctions d'un générateur de sons et ne concerne qu'un canal spécifique. *Msg Type + Channel + Controller Number + Control Value*. Il en existe toute une liste détaillée dans la spécification de MIDI 1.0.

Channel mode

Les messages *channel mode* affectent la façon dont l'instrument répond aux messages MIDI. Les messages *Omni On* et *Omni Off* indiquent si l'instrument doit traiter les informations qu'il réceptionne sur tous les canaux. Les messages *Poly On* et *Poly Off* indiquent si l'instrument doit prendre en compte plusieurs notes simultanément, s'il accepte la polyphonie.

Les system messages

Il en existe de trois types : system common, system real-time et system exclusive.

System common

Ce message concerne l'ensemble des récepteurs et active les fonctions de synchronisation, de sélection et ou de positionnement de certains équipements ne respectant pas la norme MIDI.

System real-time

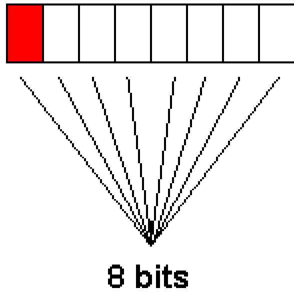
Ce message permet de synchroniser tous les équipements MIDI d'un système (séquenceur, boite à rythme). Les claviers ignorent normalement ce genre de message. Ces messages sont prioritaires sur les autres messages et peuvent apparaître n'importe où dans le flux de données (par exemple entre le byte de statut et le byte de données d'un autre message MIDI).

- Timing clock
- Start
- Continue
- Stop

System exclusive

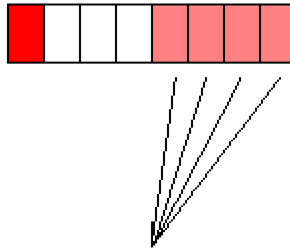
Ce sont des messages dont le contenu est laissé à l'appréciation de de chaque constructeur particulier. La norme MIDI précise juste que le message doit débuter par un numéro d'identification du constructeur et qu'un fanion particulier (EOX) indique la fin du message. La taille des messages *system exclusive* peut donc varier. La mise à jour de l'OS d'un synthétiseur peut par exemple se faire de l'ordinateur vers le synthétiseur à travers la connexion MIDI, sous la forme d'un tel message.

MIDI byte:



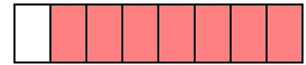
This first bit is always used to indicate whether byte is 'Status' or 'Data'

Status byte:



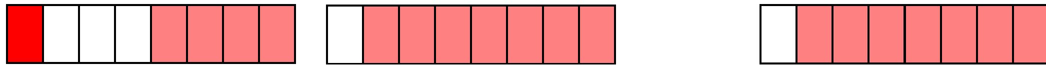
Lowest four bits used for MIDI channel number, leaving only three bits for actual instruction

Data byte:



As a data byte is always associated with the preceding Status byte, no channel number is needed, leaving seven bits free for actual data

MIDI Message:



First byte is always 'Status'

Usually followed by one or two 'Data' bytes

Figure 7 : Structure des messages MIDI

Command		Number of Data bytes	Title
Bin	Dec		
1000XXXX	128+X	2	Note Off
1001XXXX	144+X	2	Note On
1010XXXX	160+X	2	Key Pressure
1011XXXX	176+X	2	Control Change
1100XXXX	192+X	1	Program Change
1101XXXX	208+X	1	After Touch
1110XXXX	224+X	2	Pitch Bend
11110000	240	????	System Exclusive
11110sss	240+s	0 to 2	System Common
11111nnn	240+n	0	System Real Time
XXXX is the MIDI channel number. sss and nnn are switches to extend the meaning of commands.			

Table 2 : Les instructions MIDI et les bytes de données correspondants

4 Perception musicale

A la question « Que faut-il pour que les musiciens puissent interagir au travers d'un réseau ? » ; nous aurions tendance à répondre très promptement en terme de débit, de qualité de service, de gigue minimum, etc. Si ces aspects ont une influence énorme sur toute communication électronique, il faut cependant d'abord mieux identifier qui est le vrai récepteur des informations transmises, c'est à dire, dans notre cas, le musicien. Sous cet angle une réponse correcte à la question est : « Il faut qu'ils ne se rendent pas compte de l'existence du réseau entre eux ». Cette réponse simpliste évoque pourtant l'importance de considérer que le but est de rendre la transmission transparente et non pas de la rendre techniquement parfaite.

Une analogie peut-être faite avec les systèmes vidéos, en effet si la vue ne nous permet pas de saisir plus de 24 images par seconde, il existera peu d'intérêt à développer des téléviseurs capables d'afficher à des cadences plus élevées.

Ainsi, une première étape dans le développement de notre système est d'appréhender les fonctions perceptives impliquées lors du jeu. Je tiens particulièrement à remercier Mme Carolyn Drake (chercheur au CNRS CR1, Equipe Perception Auditive) et M. Bruno Repp (Senior Scientist, Haskins Laboratories) pour les informations et les réponses qu'ils m'ont très aimablement fournies dans ce domaine.

Nous allons donc ici rassembler les résultats de psychoacoustique ou de psychologie expérimentale qui nous permettrons premièrement de mieux comprendre les mécanismes mis en place lors du jeu et d'autre

part d'évaluer certains seuils perceptifs.

Le caractère important de l'interaction dans le jeu est le synchronisme qu'il nécessite. En effet les musiciens lorsqu'ils jouent à plusieurs doivent être capables de «poser» leur jeu sur une échelle de temps, un tempo qu'ils établissent. On peut donc se demander comment est-ce qu'ils procèdent pour parvenir à se synchroniser.

Il apparaît que le musicien déduit le rythme de la séquence qui précède son jeu et agit par anticipation. Cette affirmation est appuyée par certaines expériences de psychologie expérimentale. Lorsque l'on demande à un individu d'accompagner le rythme produit par un son répété à intervalle régulier, on pourra constater que l'action de l'individu va toujours légèrement précéder le rythme sur lequel il se base, ce qui montre qu'il se base sur les sons précédents.

La faculté d'anticipation peut aussi permettre au musicien de pallier aux temps de réponse présentés par un instrument. Nous entendons ici par temps de réponse, le temps écoulé entre l'activation d'une touche d'un clavier et l'apparition réelle du son généré par cette action. Nous pouvons illustrer ce phénomène par la faculté d'anticipation dont les organistes font preuve lors de la manipulation de leur instrument qui peut présenter des temps de réponse de plusieurs secondes.

L'anticipation est un phénomène naturel pour l'être humain car lui-même lorsqu'il agit se voit contraint de tenir compte de la différence de temps de réponse entre le système nerveux et le système moteur.

Il est tout aussi intéressant de pouvoir établir la précision temporelle de la perception auditive. L'oreille ménage un temps de réponse, elle distingue clairement deux événements sonores successifs à partir

d'un délai minimal de 50 ms. Au-dessous de ce seuil, la sensation glisse doucement de la perception d'un rythme à celle d'une note dont la fréquence est fixée par la cadence des répétitions. Isolé, un battement sec de tambour sera entendu comme un choc furtif. Répété 440 fois par seconde, le tambour disparaît pour laisser place à un *la* totalement virtuel, créé par nos sens comme l'est la perception du mouvement de l'image animée. Sous les 50 ms la précision va également dépendre d'autres caractéristiques. Pour un son dont l'attaque est rapide la sensibilité de l'oreille pourra être très forte, on peut dans ce cas distinguer un intervalle de 2 ms. Toutefois, en musique, on peut considérer 20 ms comme un seuil tolérable.

Cette considération est importante lorsque nous essayons de calculer la vitesse d'un jeu musical afin de contrôler les paramètres de transmission et/ou d'établir certains délais acceptables. On peut se baser pour le calcul de cette vitesse de jeu sur le nombre d'événements par unités de temps sur un intervalle. Nous obtenons alors des résultats du type 14 actions par seconde.

Imaginons cependant, que le musicien durant la durée d'un intervalle de 4 secondes joue 4 accords de 4 notes chacun, il faut donc considérer que la vitesse est de 1 événement par seconde, car rythmiquement jouer un accord correspond à 1 événement unique. Malheureusement rien ne va nous indiquer que la suite rapide de 4 notes que nous rencontrons chaque seconde est un accord, il faudra donc pouvoir grouper les événements qui se situent dans un certain intervalle et les considérer comme simultanés [A3], pour qu'automatiquement nous ne calculions pas une vitesse du jeu égale à 16 événements par seconde.

Si le réseau sur lequel nous transmettons ne délivre pas les informations d'une manière régulière il devient intéressant de savoir

dans quelle mesure cette dégradation est perceptible. Bien que le seuil de détection de gigue temporelle dans la musique ne soit pas encore déterminé, on peut estimer que la gigue est difficilement décelée si elle ne représente pas plus de 3% de l'intervalle minimum entre les notes d'une séquence. Les concepts de discrimination temporelle sont largement exposés dans les travaux de Sorkin [A4].

Il est à noter que lors de la plupart des expériences réalisées dans le domaine de la perception, toute l'attention du sujet se porte sur la détection de ce type de différence, ce qui n'est pas le cas lors de nos transmissions MIDI, donc les niveaux de tolérance peuvent généralement être un peu plus élevés que ce que de tels résultats expérimentaux suggèrent.

5 Analyse des problèmes de transport MIDI

Les messages MIDI comparativement à d'autres informations à caractère temps réel que nous pouvons transporter sur un réseau présentent l'avantage de ne demander qu'un débit limité. En revanche, l'interaction MIDI exigera des délais de transmission et de traitement relativement faibles pour rendre une impression de simultanéité. De plus nous pouvons montrer que les erreurs pouvant apparaître nécessitent souvent d'être détectées et corrigées afin de ne pas influencer la suite de la session musicale.

Afin d'établir les procédures de contrôle de flux, d'erreurs et de qualité de service que nous pouvons et/ou devons mettre en place pour un transport fiable des données sur notre réseau, il est indispensable d'analyser les problèmes que nous pouvons rencontrer lors du transfert des messages MIDI.

En première approche nous analyserons le transport des messages *Note On*, *Note Off* qui sont les plus communs lors d'une session MIDI. Nous analyserons ensuite les autres classes de messages pour lesquels les traitements seront plus au moins proches de ceux requis par ces premiers messages.

Nous analyserons les contraintes du transport MIDI au niveau de l'émetteur et du destinataire, c'est-à-dire les systèmes d'extrémités. Bien que nous nous rendrons vite compte que l'impact des systèmes intermédiaires est loin d'être négligeable nous n'essayerons à aucun moment d'avoir un quelconque contrôle à ce niveau. Ainsi tout développement ou application de ces observations se matérialiseront

sur les systèmes de bout en bout, éventuellement en fonction des systèmes intermédiaires.

Plus concrètement nous travaillons au minimum au niveau de la couche transport du modèle OSI (systèmes intermédiaires = commutateur, routeur, lien).

L'analyse de chaque cas est réalisée de manière indépendante. Au cours de cette analyse nous évaluons la manière dont le récepteur réagit sur base des informations fournies selon la norme MIDI. Les solutions qui apparaissent lors de l'analyse d'un cas problématique ne sont donc pas automatiquement reportées aux cas suivants. En conclusion de cette analyse, nous évaluerons donc les mécanismes nécessaires pour la résolution de l'ensemble des problèmes rencontrés.

5.1 Les retards

Cas 1 : Retard constant

Le meilleur cas dans lequel nous pourrions nous trouver lors de la transmission de nos messages MIDI est celui d'un retard constant très faible et une absence totale de perte ou d'erreur. Dans ce cas nous ne pouvons qu'espérer que le retard soit suffisamment faible pour permettre d'avoir une impression de simultanéité et nous pourrons alors directement transmettre le message vers le générateur de sons. On peut expliquer ce retard par le temps constant de propagation du signal à travers les lignes de transmission additionné au temps de traitement (négligeable dans la plupart des cas) à travers les différents nœuds rencontrés sur le réseau.

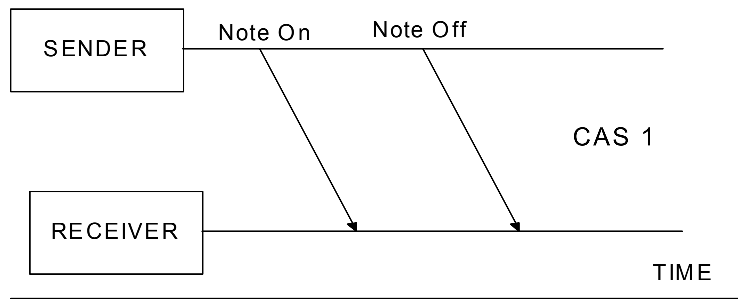


Figure 8 : Retard constant

Cas 2 : Retard variable

Imaginons une situation moins idéale mais plus proche de la réalité. Le réseau ne délivre pas tous les paquets à la même vitesse. La gigue peut subvenir à cause de la disparition ou de l'apparition de trafic externe sur les équipements du réseau que les paquets traversent ou par le changement de la route que nos paquets empruntent lors de leur acheminement. Pour que le récepteur se rende compte d'une telle situation, il faut, soit, que les paquets comportent un horodatage qui permette d'évaluer le temps de transmission, soit qu'ils soient envoyés à intervalles réguliers mêmes s'ils ne contiennent pas d'infos.

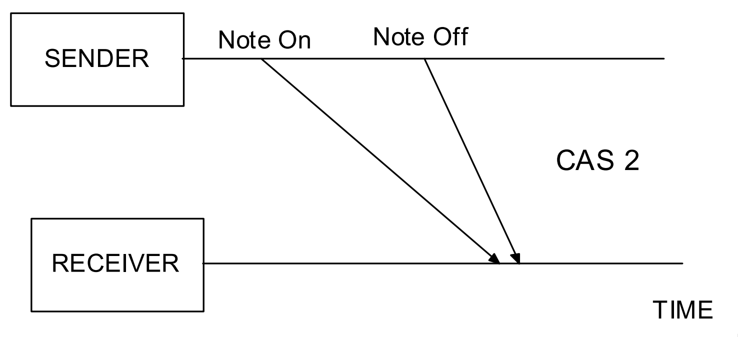


Figure 9 : Retard variable

Nous pouvons résoudre ce type de problème de deux manières :

Soit, on va systématiquement uniformiser les délais des messages

dans le temps en utilisant une mémoire tampon, ce qui en pratique consiste à augmenter le délai pour les paquets arrivant très rapidement ! Mais cela peut résoudre le problème si le délai de temporisation nécessaire reste suffisamment faible pour préserver la dynamique de réponse que les musiciens peuvent s'attendre à rencontrer. Il s'agit donc de faire un compromis entre gigue et délai moyen de réponse !

Soit, nous jugeons qu'il vaut mieux prendre en compte les notes dès qu'elles arrivent sans temporiser car le délai induit nécessaire n'est pas acceptable en regard du seuil de retard maximum à préserver pour conserver l'impression de simultanéité. Dans cette optique il reste donc encore à décider s'il faut jouer les notes tardives ou pas. Il n'est pas aussi évident de trancher qu'il y paraît.

Cas 3 : Inversion

Les Notes ne nous arrivent pas dans l'ordre d'émission, ce problème peut se présenter si le protocole que nous utilisons ne garantit pas la séquence chronologique de réception de paquet et qu'au moins deux routes différentes ont été utilisées lors de l'acheminement de nos Paquets.

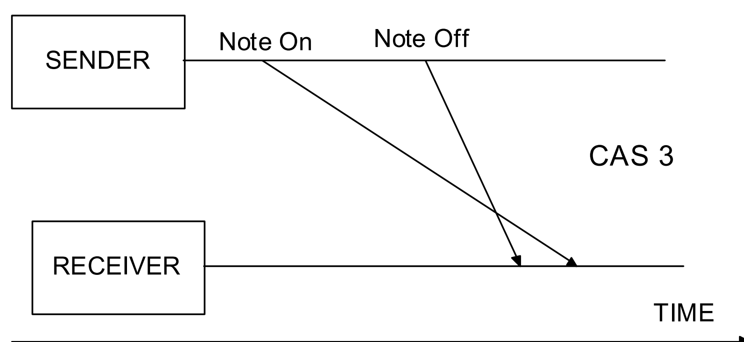


Figure 10 : Inversion de messages

Le paquet devra contenir une information de séquence ou un horodatage pour pouvoir détecter ce type de problème. Le stockage

par le récepteur d'informations sur les dernières notes reçues permettrait de détecter en partie la cohérence des notes reçues étant donné qu'après un *Note On* d'une touche on ne peut recevoir pour la même touche qu'un *Note Off* et vice-versa.

Les mêmes alternatives s'offrent à nous et il est toujours possible que le *Note On* doive être pris en compte malgré que l'indication contraire nous soit déjà parvenue. Il s'agit toujours de déterminer s'il faut jouer les notes tardives.

Dans ce troisième cas, à condition de garder trace des derniers messages reçus, nous pouvons savoir dès que nous recevons un *Note Off* si *Note On* nous a échappé et donc décider de la manière de remédier à cette situation avant même de recevoir le message manquant. Il faut toutefois qu'un horodatage ou un numéro de séquence nous permette d'être sûr qu'il ne s'agisse pas d'un ancien *Note On* retardé par le réseau.

5.2 Les pertes

La perte de paquets peut subvenir pour diverses raisons ; un équipement du réseau peut constater la corruption des données qu'il réceptionne ou subir une telle saturation qu'il n'est plus capable de procéder à l'acheminement des paquets. Les routeurs pour éviter la prolifération de paquets perdus sur le réseau peuvent également supprimer un paquet jugé trop âgé.

Nous traiterons de la perte dans l'optique où nous n'allons pas utiliser un mécanisme de retransmission sur demande (une justification précise de ce choix est exposée dans la partie « Choix du modèle TCP/IP » page 62) dans la mesure où, le temps nécessaire à

constater la perte additionné au temps nécessaire à transmettre la demande de retransmission et le délai induit par la retransmission elle-même nous mèneraient à un délai total qui ne semble pas acceptable dans notre système.

La détection de la perte va fortement dépendre de l'implémentation du protocole, mais il paraît déjà évident qu'une temporisation à elle seule ne va en aucun cas permettre de corriger notre problème dans le cas présent et donc ne sera pas suffisante puisque l'impact d'une perte peut-être très important.

Cas 1 : Perte du Note On

La perte d'un *Note On* au sens du protocole MIDI, c'est à dire si le contenu du message transmis ne comporte que les informations spécifiées par la norme MIDI, ne pourra être détectée que lors de la réception du message *Note Off* correspondant. Cette détection tardive suppose de plus que le récepteur conserve une trace des messages qu'il transmet vers le générateur afin de pouvoir à tout moment vérifier le statut actuel d'une note particulière (actif ou non-actif). Si aucun traitement n'est effectué, le fait de transmettre un message *Note Off* au générateur pour une note non-active n'aura toutefois pas d'effet.

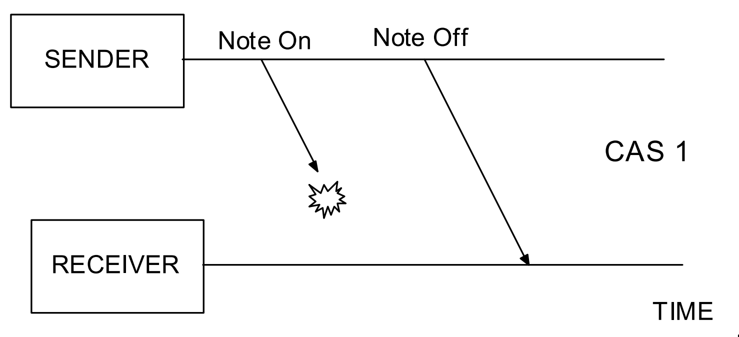


Figure 11 : Perte du message Note On

Si l'émetteur peut fournir une information supplémentaire telle qu'un numéro de séquence ou une somme de contrôle calculée d'après ce qui a déjà été envoyé, le récepteur pourra vérifier qu'il se trouve bien dans l'état normal et il sera possible de détecter tout problème de perte dès la réception d'un message contenant cette information. Il est à remarquer que selon cette approche un retard peut d'abord être considéré comme une perte par le récepteur. Cette information supplémentaire pourra être envoyée à intervalle régulier et/ou transportée par les éventuels autres messages véhiculés (autres notes sur le même ou un autre canal). Rappelons encore qu'il sera parfois nécessaire de décider s'il faut jouer une note qui n'est plus en statut On.

Cas 2 : Perte du Note Off

Ce cas nous montre un problème symétrique au cas précédent, il illustre la faiblesse d'une solution consistant à ne pas réagir aux pertes, en effet dans ce cas l'action du générateur va se poursuivre et le son produit restera audible jusqu'à ce qu'éventuellement une autre note identique vienne activer puis désactiver le générateur.

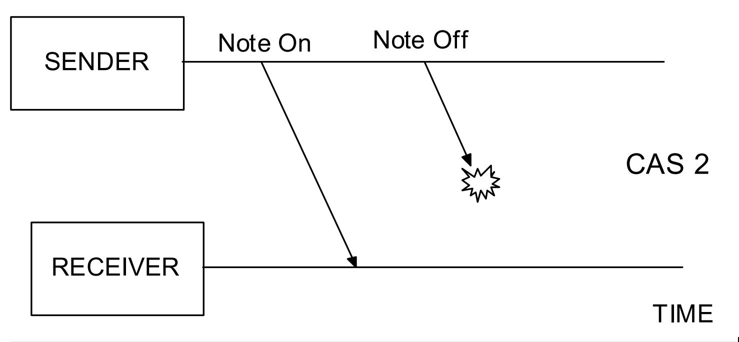


Figure 12 : Perte du message Note Off

Ce cas est particulièrement intéressant car il nous montre comment une erreur peut influencer toute la suite d'une session musicale ou pour le moins une partie non négligeable. Il faudra donc

obligatoirement avoir une information qui permette de vérifier au moins à intervalles réguliers que le générateur est dans un état conforme aux attentes.

Cas 3 : Perte du Note On et du Note Off correspondant

Ce cas illustre la perte possible de la *Note On* et de la *Note Off* correspondante.
Les observations faites pour les cas 1 et 2 nous permettent de conclure immédiatement que si un numéro de séquence permet de détecter un problème, seule une forme de redondance régulière va nous permettre d'identifier clairement le problème. En effet, sans informations supplémentaires nous ne pouvons pas déduire quel était le contenu du message perdu.

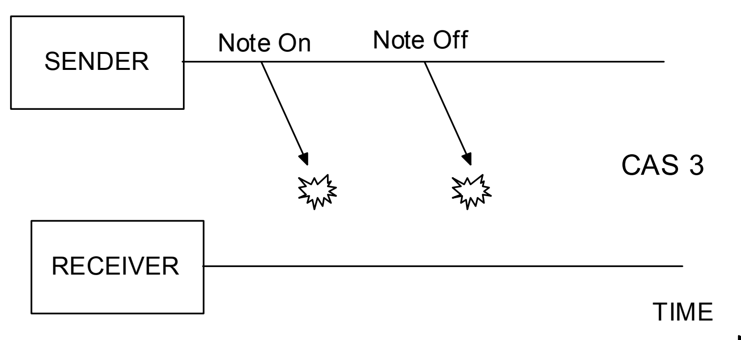


Figure 13 : Perte du message Note On et du message Note Off correspondant

5.3 Cas de perte et de retard plus ambigus

Les cas 1 et 2 nous montrent la possibilité existante de confondre différentes informations se rapportant à une même note MIDI quand le service rendu par le réseau n'est pas fiable.

Cas 1 : Inversion

Ce cas met en évidence le problème qui peut surgir si les différentes routes empruntées par les informations transmises entraînent une inversion des messages. Le *Note On* 2 va être prématurément annihilé par le *Note Off* 1 en retard. La solution qui consiste à avoir un horodatage dans les messages peut résoudre le problème. Si la gigue reste faible on peut imaginer résoudre ces problèmes par une temporisation chez le récepteur.

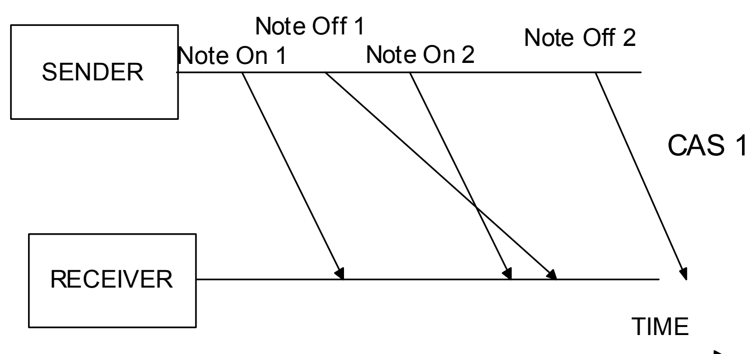


Figure 14 : Ambiguïté lors de l'inversion de messages

Cas 2 : Perte du Note On et du Note Off correspondant

(Figure 15) Le deuxième cas nous montre, hélas, que l'horodatage ne sera pas suffisant, en effet si les seules informations reçues sont le *Note On* 1 et le *Note Off* 2, l'horodatage ne permettra pas de détecter qu'une erreur s'est glissée, pas plus d'ailleurs qu'une table contenant les Notes actives dans le générateur. Il faut donc impérativement fournir une information permettant de détecter précisément la perte de données.

Il nous faut donc soit fournir un numéro de séquence à chaque message, soit avoir une information supplémentaire (somme de contrôle, redondance) qui va permettre de vérifier l'absence d'erreur.

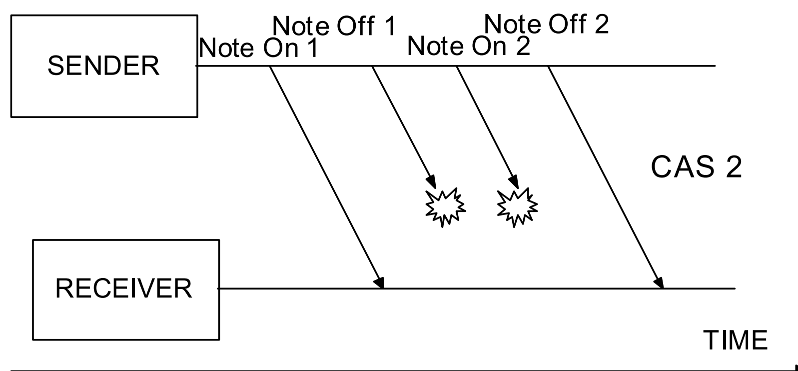


Figure 15 : Ambiguïté lors de la perte de messages

5.4 Transport d’autres types de message

Analysons maintenant les précautions à prendre dans le traitement d’autres messages :

Le message *program change* indique que sur un canal spécifié un changement de son a été introduit. Nous ne devons pas permettre qu’un son soit généré avec le mauvais programme. Nous veillerons à mettre en place un mécanisme permettant d’éviter les erreurs de ce type. Le traitement pour les *control change* sera analogue.

Les messages *SysEx* sont des messages qui ne rentrent pas vraiment dans le jeu musical, ils peuvent donc être traités avec des mécanismes d’accusés de réception et de retransmission similaires à ceux mis en place par TCP/IP.

Les messages *Pedal On/Off* concernent l’activation de la pédale qui permet de maintenir actifs, à l’aide du pied, les sons joués après son activation et jusqu'à son relâchement, elle retarde donc l’effet du relâchement de touche, transmis par un *Note Off*. Il est donc plus

prudent que le récepteur lui même cesse de retransmettre les messages *Note Off* quand la pédale est activée afin d'éviter, qu'un *Note Off* ne soit reçu avant le message *Pedal On* et que le son soit donc désactivé par le générateur. Il ne faut, lors du relâchement de la pédale, que désactiver les notes pour lesquelles un *Note Off* a été reçu durant l'activation de la pédale.

5.5 Conclusion de l'analyse des problèmes de transport

Résumons les observations que l'analyse des différents scénarios nous a fournie et choisissons les mécanismes à mettre en place pour faire face aux problèmes rencontrés.

Une mémoire tampon peut être utilisée pour autant que les temps de transmission et de temporisation (des 2 cotés) restent acceptables. Elle va permettre de supprimer la gigue de faible amplitude.

Dans un réseau non fiable l'usage de l'horodatage est indispensable pour évaluer le retard, il n'est cependant pas suffisant à lui seul car il ne permet pas la détection des erreurs. Les numéros de séquence et/ou les sommes de contrôle peuvent apporter une solution à ce type de problème qui nécessite réellement d'être pris en compte de par la dégradation importante qu'une unique perte est susceptible d'occasionner sur le jeu musical.

Il faut réfléchir au mécanisme à adopter pour permettre au récepteur de vérifier si l'état dans lequel se trouve le système est conforme. Ces considérations seront d'autant plus importantes que les pertes de plusieurs paquets successifs sont possibles (perte en rafale) et que la remise à jour en cas de décrochage du récepteur doit être rapide.

Nous avons constaté que MIDI par lui-même n'est pas suffisamment riche pour nous permettre de juger de l'action à entreprendre en cas de perte ou de retard. Nous n'avons pas une connaissance suffisante du comportement du générateur, ce qui pose un problème quant à notre façon de prendre les décisions les plus adaptées dans certaines situations.

Bien que nous ayons principalement traité de la transmission sur un seul canal, les concepts se reportent facilement aux cas où plusieurs canaux MIDI sont utilisés. Il est à noter que logiquement l'émission ne met en jeu qu'un seul canal alors que la réception peut impliquer plusieurs canaux quand plusieurs instrumentistes sont impliqués dans la communication.

6 Intervalle de transmission

L'intervalle de transmission est le temps écoulé entre l'envoi de deux paquets successifs sur le réseau.

Il faut choisir un intervalle de transmission suffisamment petit pour garder un bon rendu musical et suffisamment grand pour réaliser l'économie en termes de bande passante.

Evidemment comme on ne peut pas insérer un message dans un paquet qui est déjà parti, l'instant d'émission d'une information est par défaut un plafond relativement à l'instant d'apparition de l'événement concerné, il faut à nouveau être très prudent car il s'agit d'un délai supplémentaire. Ce qui nous entraîne à considérer la somme :

(Moment de l'envoi du paquet – Moment de l'événement) + Temps de transmission + Eventuel temps de temporisation chez le récepteur. Ce délai devra rester acceptable pour que la structure temporelle du jeu soit respectée.

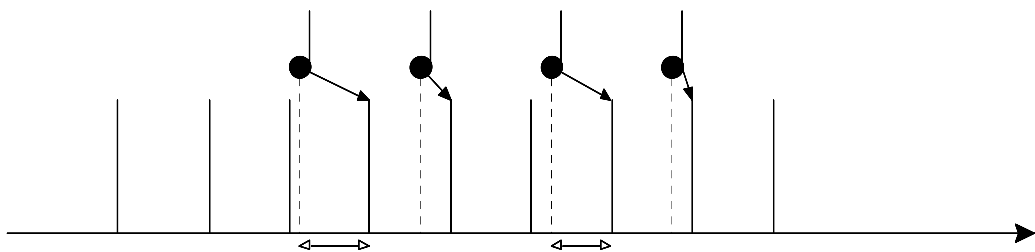


Figure 16 : Intervalle de transmission

La Figure 16 illustre la situation dans laquelle le délai avant

transmission d'un événement se rapproche de l'intervalle de transmission lui-même.

On peut se demander s'il y a un intérêt à envoyer des paquets à intervalles réguliers au risque qu'ils ne contiennent pas d'informations utiles pour le récepteur.

Si on le fait, le numéro de séquence peut remplacer efficacement l'horodatage. Temps = (N° de séquence * Durée d'intervalle)

Si on ne le fait pas, il faut soit l'horodatage pour se réaligner, soit indiquer le nombre de saut de paquet effectué relativement à une fréquence d'envoi supposée. Il faudra dans ce cas garder en plus le numéro de séquence afin de savoir si on a un saut de paquet dû à une perte ou dû à l'absence d'information à transmettre.

Voyons si l'ajout d'un horodatage dans chaque paquet de données réelles est plus lourd que l'envoi systématique de paquets

- SC = surcharge dans les couches inférieures en bytes.
- H = Nombre de byte pour l'horodatage ou le saut de paquet.
- D = taille des données utiles dans le paquet.
- TP = Nombre de bytes de toutes les autres infos du paquet.
- N = Nombre de paquet contenant réellement des données par unité de temps.
- F = Nombre de paquets envoyés par unité de temps quand on procède à l'envoi régulier.

Si l'envoi est réalisé sur base régulière, nous obtenons un débit en byte par unité de temps = $(F – N) * (SC + TP) + N (SC + TP + D)$

Si l’envoi est réalisé seulement quand l’info est présente. Envoi avec horodatage, nous obtenons un débit en byte = N * (SC + TP + H + D)

MIDI nécessite un intervalle de transmission relativement court pour pouvoir quantifier temporellement les paquets avec suffisamment de précision. Mais contrairement à la situation en audio, il n’y a pas souvent de paquet à émettre relativement à la précision que nous souhaitons atteindre. Donc F doit être beaucoup plus grand que N. Nous avions vu (page 27) que la gigue temporelle n’est pas décelée si elle ne représente que 3% de l’intervalle minimum entre les notes. Choisissons cependant une valeur minimale pour F = 10 N.

SC, la surcharge à laquelle IP participe, est importante, nous pouvons donc considérer que la transmission systématique à intervalle régulier ne sera pas intéressante. Si nous imaginons une transmission par le protocole RTP (*Real Time Protocol*) [A6] au-dessus de UDP (*User Datagram Protocol*), la surcharge due à l’encapsulation par les couches inférieures nous donne déjà à elle seule une valeur minimum de surcharge de 40 bytes (IP+UDP+RTP). (voir 10.3 Choix du modèle TCP/IP).

Nous estimons que les données elles-mêmes peuvent être représentées par 20 Bytes et l’horodatage peut être codé avec 4 Bytes.

Le rapport entre l’envoi régulier et l’envoi uniquement quand les données sont disponibles nous donne alors

$$(9 * (40 + TP) + (40 + TP + 20)) / (40 + TP + 4 + 20) = 420 + 10TP / 64 + TP$$

Donc, dans notre cas l’envoi régulier est loin d’offrir un service intéressant. Il faut cependant nuancer cette affirmation, car une

compression de la pile IP+UDP+RTP existe mais n'est applicable que pour les liaisons point a point. [A7]

Si l'utilisation d'un intervalle de transmission constant n'est pas adaptée, un petit intervalle de temps entre la réception d'information et la transmission permet de regrouper les événements quasi-simultanés pour une transmission unique. Imaginons que le musicien joue un accord de par exemple 5 notes. Lors de l'arrivée de la première note à l'émetteur de paquet, on peut attendre un temps restreint pour récupérer les autres notes activées simultanément. Dans ce cas, on peut transmettre les différentes notes dans le même paquet et ainsi épargner la surcharge due à l'encapsulation. Une estimation du seuil temporel minimal pour que les événements soit perçus comme simultanés à été donné dans la partie 4 : Perception musicale.

7 Pré-Analyse du comportement MIDI

Nous avons pu constater à plusieurs reprises que le manque d'informations concernant le comportement du générateur de sons nous posait des problèmes quant à la prise de décision dans les situations complexes.

La question est de savoir d'une part si des informations supplémentaires peuvent nous être utiles et d'autre part si l'usage d'informations qui ne sont pas à la base disponibles va rendre notre procédé difficile à mettre en œuvre dans une installation MIDI classique.

Commençons par voir comment le générateur produit les sons en fonction des informations MIDI qu'il reçoit. Les instruments de musiques MIDI génèrent leurs sons en faisant varier l'amplitude du signal émis selon une enveloppe prédéterminée (voir Figure 17).

La production sonore passe par généralement par 4 étapes :

- Attack
- Decay
- Sustain
- Release

Chaque son a une caractéristique ADSR (*Attack, Decay, Sustain, Release*) différente, l'attaque d'un violon par exemple sera beaucoup plus progressive que celle d'un piano. Une forte modification des caractéristiques ADSR empêche un instrument d'être reconnu.

L’amplitude de l’enveloppe elle-même dépendra de la pression exercée lors de l’activation de la touche.

Bien que l’émetteur connaisse le temps entre le début de l’*attack* et la fin du *sustain* puisque qu’il retransmet ses informations, il ne dispose pas au départ d’informations sur la durée des différentes phases et leurs amplitudes. Après le *Note Off*, le générateur passe en phase de relâchement et à ce niveau aucune indication ne nous permet de savoir à quel moment le son est réellement éteint.

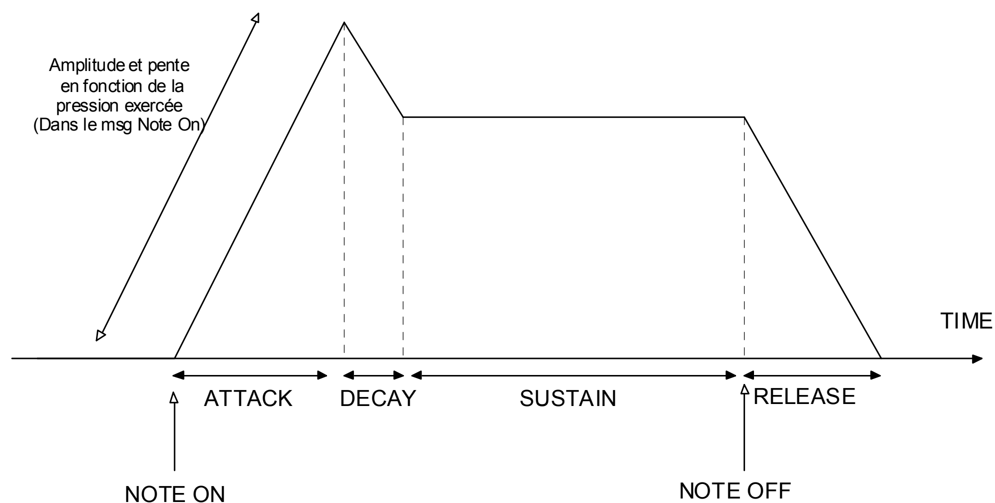


Figure 17 : Enveloppe ADSR

Peut-on imaginer fournir à notre système de communication les caractéristiques ADSR ? Pour plusieurs raisons nous pouvons montrer que cette hypothèse n’est pas très contraignante :

Les différents instruments du modèle Général MIDI partagent les mêmes caractéristiques.
Certains instruments de musique évolués permettent d’accéder à ces informations afin de modifier les caractéristiques des sons au souhait

de l'utilisateur. Nous pouvons également, le cas échéant, supposer que de telles caractéristiques se retrouvent dans les documents de spécifications de l'instrument.
S'il n'est cependant pas possible d'accéder à ces informations, un utilitaire capable de procéder à l'analyse automatique de ces caractéristiques peut facilement être mis au point.
Les caractéristiques ADSR sont utiles pour la correction d'informations après transmission, donc chez le récepteur ; le musicien ne doit donc posséder les caractéristiques ADSR que de son propre instrument.

Essayons de montrer à travers deux cas la façon dont nous pouvons tirer avantage de cette connaissance.

Revenons à un cas que l'on a déjà vu (Figure 18) :
Nous recevons une indication *Note On* très tardivement, quasiment au même moment que l'indication *Note Off* qui indique que le générateur doit passer en phase de release.

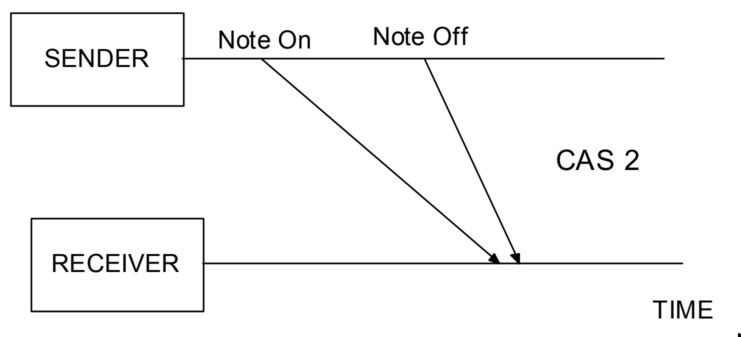


Figure 18 : Retard variable

Le choix de jouer ou d'abandonner une note en retard est très difficile. Il faut en effet tenir compte qu'on risque dans un cas comme dans l'autre d'altérer la structure du jeu ; que ce soit sa structure mélodique ou harmonique, si on ne joue pas la note ; ou sa structure temporelle

(rythmique), si on la joue.

Si systématiquement, nous abandonnons les notes tardives, nous manquons aussi toute la partie du son qui devait apparaître après le *Note Off*, si le release est très long cela transforme un léger retard en une perte ayant une incidence sur un intervalle beaucoup plus large que celui qui existe entre le *Note On* et le *Note Off*. (Figure 19)

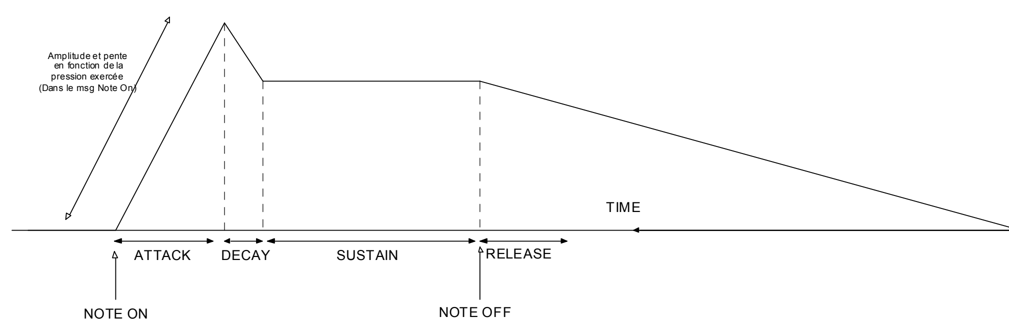


Figure 19 : ADSR avec relâchement long

L'éventualité de jouer toutes les notes en retard peut également poser des problèmes. Effectivement le retard subi en rapport à la durée complète de l'incidence sonore, c'est à dire le son résultant avec la période de *release*, peut être trop important pour qu'il soit nécessaire d'activer le son. (voir Figure 20)

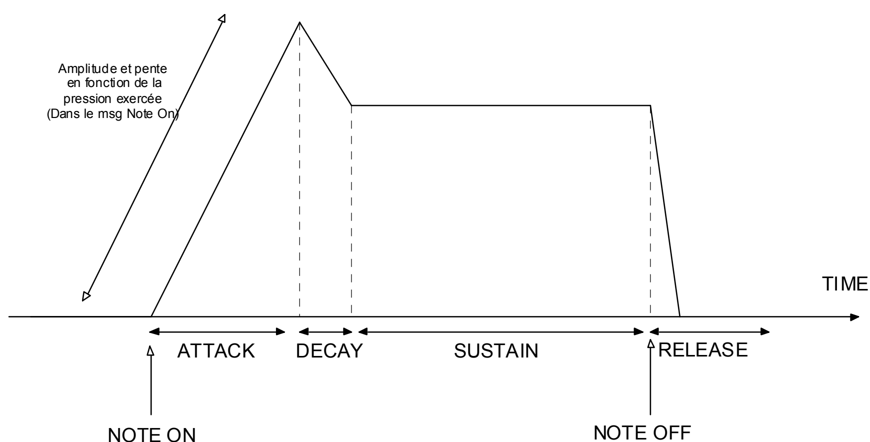


Figure 20 : ADSR avec relâchement court

Si on ne peut pas considérer la durée du *release* comme un critère complet pour la prise de décision en cas de retard, on peut toutefois estimer qu'il apporte une information utile.

Nous avons vu que l'*attack* aura une influence sur le seuil de perception temporelle de l'apparition des notes. On pourra donc également utiliser cette caractéristique pour prendre certaines décisions.

Instrument type	Attack time	Release time
Electric organs	1,5 to 4,6 msec	14 to 35 msec
Pipe organs	14 to 35 msec	35 to 170 msec
Reed instruments	35 msec	35 to 75 msec
Bowed instruments	14 to 95 msec	170 msec
Percussive instruments	1,2 msec	170 msec to several seconds
Plucked instruments	1,2 msec	14 msec to several seconds

Table 3 : Temps d'attaque et de relâchement pour différents types d'instrument

8 Vérification de l’état du système

On se rend très vite compte que la perte d’une note peut avoir des conséquences importantes. Etant donné que la demande de retransmission ne semble pas très adéquate à cause du temps minimal qu’elle requière (Délai de détection d’un problème + délai de transmission de la demande de retransmission + délai de retransmission de la demande …) Il semble plus approprié que l’émetteur lui-même puisse, à certains intervalles, envoyer les informations nécessaires pour que le récepteur soit capable de se remettre dans un état cohérent après une perte d’information.

Les adaptations du protocole MIDI à la transmission fonctionnent généralement de deux manières :

- Les indications *Note On* et *Note Off* sont retransmises à maintes reprises. Evidemment cette solution pose un problème de surcharge de la bande passante. Car pratiquement il faut retransmettre constamment le statut de tout le «clavier». [A2]

- Une autre approche [A8] consiste à ne renvoyer, sous forme de journal, que les événements qui se sont produits depuis un instant clé, défini par l’analyse de rapports que les récepteurs émettent pour informer l’émetteur de la qualité de transmission actuelle. Un problème peut se poser si on émet vers beaucoup de récepteurs, car la taille du journal va dépendre de la capacité de tous les récepteurs à fournir un accusé de réception rapidement.

En travaillant avec un codage différent de celui utilisé lors de la

transmission des messages *Note On* ou *Off* individuels, nous choisissons de transmettre une vue complète de l'état cohérent dans lequel le système récepteur doit se trouver et à moindre coût. L'idée est de transmettre simplement une table contenant le statut binaire de chaque note. Accordons-nous sur le fait qu'un bit à 1 signifie que la note est active (On) et un bit à 0 la note ne l'est pas (Off).

Chaque octave est constituée de 12 notes, si nous comptons les 10 octaves possibles, ce qui est très large, nous pouvons donc ainsi transmettre un état sommaire du clavier à intervalle régulier, cette solution nous demandera donc 120 Bits (15 Bytes). Nous pourrions réduire ce nombre en segmentant le clavier en octave et en ne transmettant que les octaves sur lesquelles le jeu prend place.

Les informations sont légèrement tronquées puisque la pression exercée n'est pas précisée pour chaque note, mais nous estimons que le récepteur peut calculer une valeur acceptable en fonction des dernières notes qu'il a reçues ou que l'émetteur lui-même peut calculer une valeur moyenne pour les notes actives qu'il émet.

Il pourrait également être intéressant de déduire l'intervalle à utiliser pour la transmission de cet état à partir de la vitesse du jeu exécuté.

Si nous pouvons inclure cet état dans les messages normalement véhiculés, il faut toutefois fournir cette table quand l'intervalle entre 2 messages devient trop important. Imaginons en effet que le dernier message soit perdu, tant qu'une nouvelle note n'est pas activée, l'émetteur ne reçoit pas d'information lui permettant de se remettre à jour. Il faut donc transmettre l'état individuellement, si aucun changement n'a lieu durant une trop longue période et éventuellement répéter l'opération à certains intervalles (de préférence croissant) pour pallier la perte de l'état lui-même.

9 Mécanisme d’anticipation

Alors que la gigue et les pertes sont surtout gênantes dans la mesure où elles dégradent l’information musicale transmise, le délai de transmission sur le réseau, ne pose pas le même type de problème.
Un délai important est inacceptable principalement car il empêche les musiciens de jouer de manière synchrone. Etudions la possibilité de mettre en place un mécanisme permettant de limiter l’impact du retard que subissent les informations musicales lors de leur transmission sur la faculté de synchronisation des musiciens. Nous avons vu précédemment que le mécanisme humain de synchronisation musicale consistait à extraire le tempo de la séquence musicale qui précède l’action même du musicien (voir partie 4 Perception musicale).

Analysons, le cas d’un musicien B qui tente d’accompagner le musicien A situé à l’extrémité opposée d’un réseau. Pour la clarté de l’exemple, imaginons que A veuille jouer la même chose que B et au même moment. Avant de jouer B va donc analyser le jeu de A, il va en extraire un tempo et ensuite il va, sur base de ce tempo, reproduire la séquence précédemment jouée par A. Le jeu de A parvient à B avec un délai de x ms. B va calculer son anticipation pour pouvoir jouer lui-même avec ce retard, soit x ms trop tard. A reçoit le jeu de B y ms plus tard.
Si le délai est important la réponse de B à A sera gênante pour celui-ci.

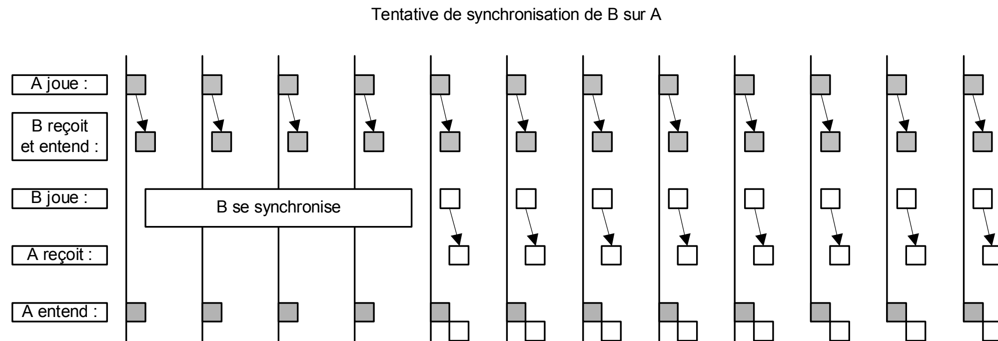


Figure 21 : Transmission avec temps de réponse nul au niveau du clavier

Le problème est dû au délai de transmission qui conduit à un retard entre le moment où A et B entendent le son (voir Figure 21). Imaginons maintenant que les claviers de A et de B présentent des temps de réponse égaux respectivement au délai de transmission de A vers B et de B vers A mais que la transmission s'opère dès l'activation. (voir Figure 22), Les notes jouées sont donc restituées au même moment chez A et chez B. En fait B va anticiper pour pallier au temps de réponse de son propre instrument et va donc jouer en même temps que A. Les 2 notes vont se croiser sur le réseau.

On peut donc imaginer absorber l'impact du délai de transmission sur le réseau en augmentant le temps de réponse pour la restitution locale (voir Figure 23). Comme nous l'avons expliqué, dans la partie traitant de la perception musicale, la capacité d'anticipation de l'acteur humain permet de s'accommoder d'une différence entre l'instant où il effectue l'action et l'instant où la réaction se produit, pour autant que ce temps de réponse reste raisonnable.

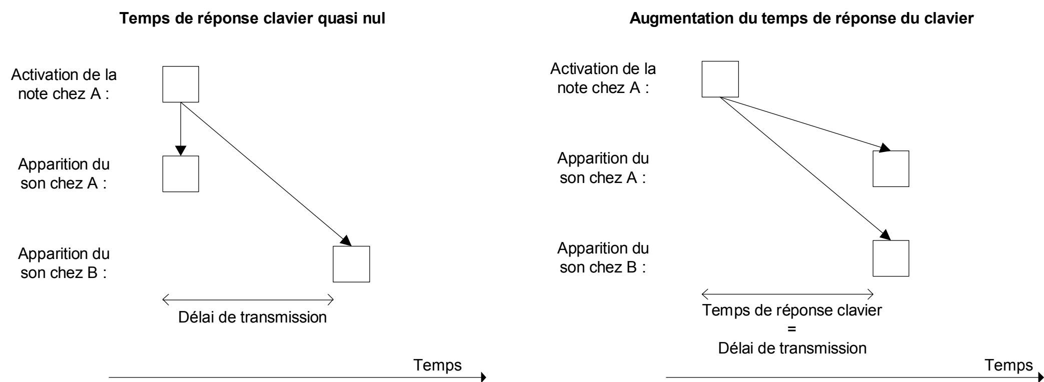


Figure 22 : Modification du temps de réponse

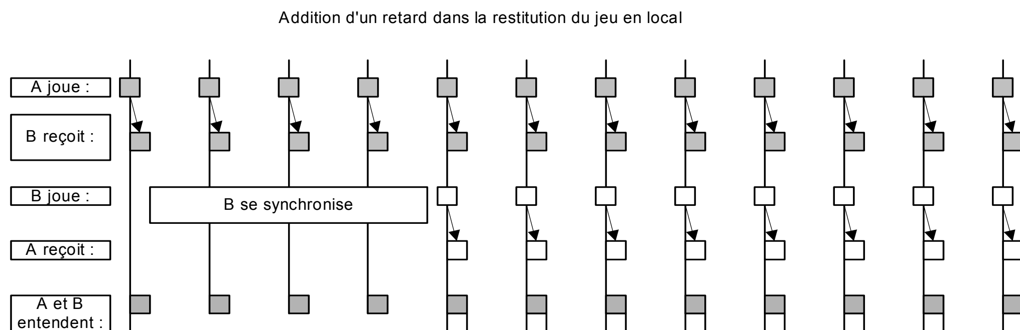


Figure 23 : Adaptation du temps de réponse

Cette démarche a toutefois ses limites car il est évident que le synchronisme se gagne ici au dépend du temps de réponse de l’instrument. Dans un but d’illustration nous avons donc réalisé des tests permettant de mesurer le niveau de perception du délai induit en fonction de la vitesse du jeu.

Nous avons, pour une série de temps de réponse, joué une suite d’accords, un sur chaque temps indiqué par un métronome réglé à la vitesse indiquée en BPM (Beats per Minute). Nous notons pour chaque combinaison temps de réponse/BPM le degré avec lequel le temps de réponse est perçu (voir Table 4 : Niveau de perception en fonction du temps de réponse et du tempo

Temps de réponse	BPM = 60	BPM = 90	BPM =130	BPM =160	BPM = 200
0 ms	Non	Non	Non	Non	Non
20 ms	Non	Non	Non	Non	Léger
40 ms	Non	Non	Léger	Léger	Fort
60 ms	Léger	Léger	Oui	Fort	Fort
80 ms	Léger	Oui	Fort	Fort	Fort
100 ms	Oui	Fort	Fort	Fort	Fort
150ms	Fort	Fort	Fort	Fort	Fort
200 ms	Fort	Fort	Fort	Fort	Fort

Table 4 : Niveau de perception en fonction du temps de réponse et du tempo

Bien que la validité de ces résultats soit très subjective, étant donné que l'estimation n'a été réalisée que par un sujet, on peut toutefois mettre en évidence qu'il est très difficile de détecter le délai induit avant qu'il atteigne une valeur de 40 ms et surtout que le délai induit devient de plus en plus perceptible quand le jeu s'accélère.

La mise en place d'un tel mécanisme pourrait donc tirer avantage d'une caractéristique dynamique permettant d'adapter le délai de réponse de l'instrument à la vitesse réelle du jeu.

10 Choix de l’architecture réseaux

10.1 Objectifs et contraintes

Nous devons faire le choix de l’architecture réseau en tenant compte aussi bien de nos objectifs que des contraintes qui sont inhérentes au transport du MIDI.
Nous pouvons résumer en quelques points les caractéristiques qui nous semblent indispensables pour mettre en œuvre notre système.

Nous souhaitons que le système puisse fonctionner sur les réseaux les plus communs et puisse profiter de l’Internet pour permettre d’effectuer des connexions à distance vers n’importe quel autre équipement qui y serait connecté.

Il est nécessaire que le délai moyen de transmission reste relativement faible. Une valeur de 200 ms semble un maximum au-delà duquel aucune transmission MIDI ne peut être qualifiée d’interactive.

Il est souhaitable que la gigue soit suffisamment faible pour permettre une correction efficace.

Avant de procéder à nos propres choix, il est intéressant d’illustrer ici la technologie mise au point par Yamaha. Nous ne manquerons pas de constater qu’il s’agit d’une réponse à la problématique que nous avons soulevée dans ce que nous avons appelé le modèle Studio, ce qui tend à conforter le fait qu’il s’agisse d’un besoin réel dans le contexte professionnel.

10.2 Yamaha mLAN

La technologie mLAN (*music Local Area Network*) de Yamaha [A9, A10] utilise le protocole IEEE1394 inventé par Apple. IEEE1394 est une connexion bus série bon marché, mais puissante qui permet des transmissions temps réel très performantes.

Le mLAN a été développé par Yamaha afin de supporter plusieurs canaux de communications audionumériques et MIDI et permettre une configuration aisée des différents périphériques compatibles.

Le mLAN permet le transfert des données audio et musique à une vitesse de 200 Mbps. Cela représente approximativement jusqu'à 100 canaux d'audionumérique ou 256 connexions MIDI.

Contrairement aux systèmes conventionnels le mLAN permet de véhiculer simultanément sur les mêmes câbles aussi bien les données Audio que MIDI ce qui engendre une simplification appréciable de la connectique dans les studios.

La technologie mLAN permet le routage des informations audio/musique et autorise donc la mise en place d'un réseau musicale apte à répondre à des attentes différentes sans que des changements de connexions soient nécessaires. La possibilité de connecter les équipements à chaud sur le réseau (*hot pluggable*) rend encore plus facile l'ajout d'un nœud.

Les équipements mLAN sont aussi bien capables de gérer eux-mêmes leurs connexions que de se soumettre à la coordination d'un gestionnaire de connexion matérialisé soit par un équipement

spécifique, soit par un ordinateur doté du logiciel de gestion approprié et de l'interface IEEE1394.
L'interface IEEE1394 (FireWire ou i.Link chez Sony) se retrouve actuellement en standard sur nombre de PC grand public grâce au succès du grand nombre de périphériques numériques qui l'exploitent avantageusement.

On peut donc s'attendre au succès de cette technologie dans le monde des ingénieurs du son, d'autant plus qu'un autre grand constructeur d'instrument de musique électronique, KORG, adopte lui aussi le standard mLAN sur ses modèles haut de gamme.

Le protocole mLAN autorise deux modes de transfert, premièrement le mode synchrone adapté pour les informations nécessitant une transmission prioritaire et deuxièmement le mode asynchrone pour les transmissions standard. (voir Figure 24 : Pile protocolaire mLAN)

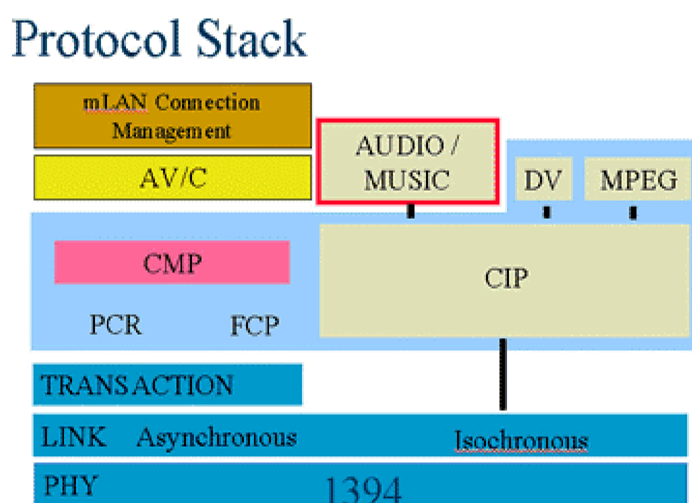


Figure 24 : Pile protocolaire mLAN

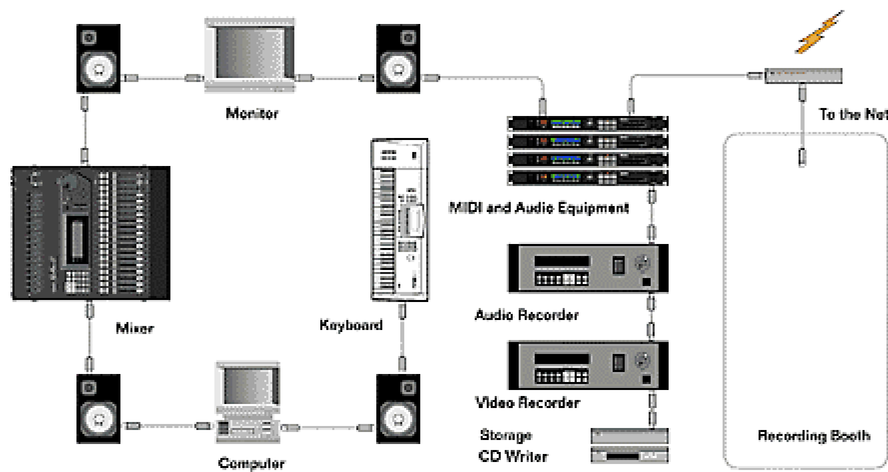


Figure 25 : Réseau mLAN

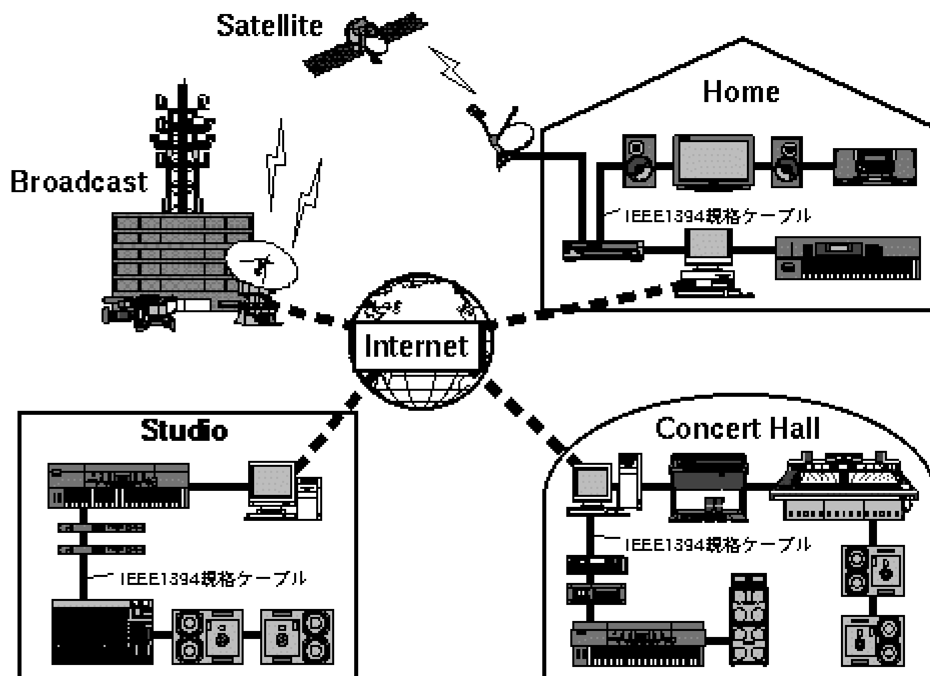


Figure 26 : Connectivité de réseau mLAN

Il sera intéressant de suivre l'évolution de cette technologie qui promet régulièrement des débits plus importants et une envergure de réseau plus étendue.

On pourra toutefois déplorer que l'absence de support direct du protocole TCP/IP empêche l'adressage direct des différents équipements et par conséquent la transparence de l'infrastructure utilisée. Un équipement spécifique est toujours nécessaire pour effectuer la traduction de l'adresse IP en un adressage propre à mLAN. (voir Figure 26)

L'adaptation du protocole IP au-dessus de IEEE 1394 [A11] apportera peut-être enfin la possibilité de réellement étendre les réseaux ainsi formés à l'échelle du Web.

10.3 Choix du modèle TCP/IP

Il est clair que nous aurions pu nous baser sur le modèle de référence OSI afin d'étudier le choix d'architecture de manière plus impartiale, cependant, se fixer comme objectif essentiel de permettre la communication sur l'architecture de réseau la plus répandue nous impose le choix du modèle de référence TCP/IP.

Pour rappel TCP/IP, du nom des ses deux principaux protocoles, est surtout né du besoin de relier des réseaux très divers de la manière la plus transparente possible. L'accent à été mis sur la disponibilité du réseau qui devait permettre à l'ordinateur émetteur de continuer à émettre vers l'ordinateur récepteur en cas de panne de certains systèmes intermédiaires, le besoin d'une architecture souple était essentiel puisque l'on voulait disposer d'applications aussi différentes

que le transfert de fichiers et la transmission de la parole en temps réel. [L1]

Ces desiderata ont conduit à choisir un réseau à commutation par paquets fondé sur une couche d'interconnexion de réseaux sans connexions.

Application
Transport
Internet
Host-Network

Figure 27 : Pile protocolaire TCP/IP

Pour au moins deux raisons, nous ne procéderons pas à l'analyse du choix des composantes de la couche la plus basse du modèle TCP/IP. En effet, d'une part il semble évident qu'un trop grand nombre de considérations doivent être prises en compte pour évaluer de manière sérieuse les différentes alternatives, d'autre part, l'optique de ce mémoire est toujours de mettre en place un système permettant dans la mesure du possible de pallier aux défauts induits par les conditions réseaux sous-jacentes et de prendre en compte la diversité des connexions sur lesquelles une communication MIDI peut transiter. Nous commencerons donc notre analyse au niveau de la couche Internet.

La couche Internet

La couche Internet est primordiale, elle permet de procéder à l'acheminement des paquets indépendamment les uns des autres, de, et vers, tout type de réseaux. Elle définit un format officiel de paquet et un protocole qu'on appelle IP (*Internet Protocol*). Le rôle de la couche Internet est de remettre les paquets IP à qui de droit. Le

routage des paquets est donc une fonction primordiale de cette couche. L'adressage est également défini au niveau de cette couche. Les caractéristiques importantes à prendre en compte au-niveau de cette couche sont les suivants :

IP se base sur le principe du meilleur effort, ce qui signifie en pratique qu'au niveau de cette couche, le transfert est non fiable, la remise des paquets n'est pas garantie, un paquet peut être perdu, dupliqué ou remis hors séquence sans que rien n'en informe l'émetteur ou le récepteur.

Les paquets peuvent utiliser des routes différentes, ce qui va induire une gigue possible qui risque de mettre en péril la qualité de la communication que nous souhaitons mettre en place.

Parmi les champs du datagramme IP (Figure 28) se trouve un champ Type de service (TOS) qui pratiquement n'est pas souvent pris en compte, mais qui pourrait permettre de spécifier une qualité de service à fournir selon les besoins. (voir partie 14 : QoS)

Nous ne détaillerons pas ici le choix des protocoles de routage, car nous faisons toujours l'hypothèse que nous ne maîtrisons pas le routage des informations que nous envoyons sur le réseau. Bien sur, si nous faisions l'hypothèse que nous travaillons sur un réseau privé plus important doté de routeurs, il serait intéressant d'étudier la question du choix de l'algorithme de routage de paquets approprié.

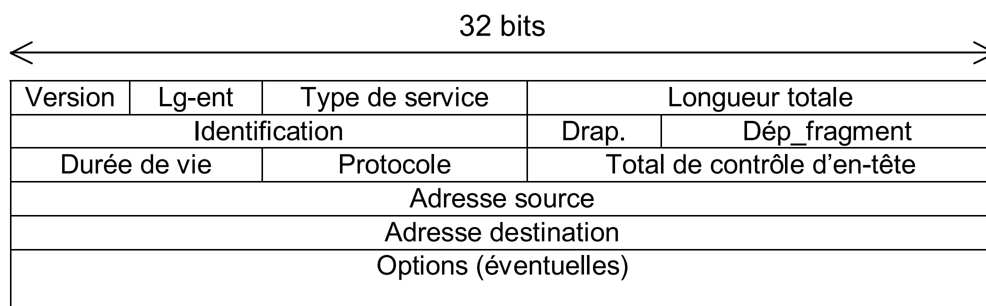


Figure 28 : Format de l'en-tête du datagramme IP

La couche transport

La couche transport se situe au-dessus de la couche internet. Elle permet à des entités paires sur des ordinateurs sources et destinations d'échanger des données. Le modèle TCP/IP propose deux protocoles pour la couche transport, TCP et UDP. Au premier abord TCP semble un candidat idéal puisqu'il fournit un service de transport fiable au-dessus de la couche IP qui n'assumait pas elle-même une telle fonctionnalité.

Pour assumer sa fiabilité TCP travaille avec des accusés de réception, quand un paquet est reçu par le destinataire, celui-ci va envoyer un accusé de réception qui indique à l'émetteur qu'il peut envoyer le paquet suivant. Si par contre l'émetteur ne reçoit pas cet accusé de réception dans un certain laps de temps, il sait qu'il doit retransmettre le paquet.

Pour que l'attente de l'accusé de réception n'interrompe pas la communication, TCP utilise un mécanisme de fenêtre glissante. Une fenêtre symbolise un certain nombre de paquets que l'émetteur peut envoyer successivement sans attente d'accusés de réception. Lorsque l'émetteur obtient la confirmation que le récepteur a correctement reçu n paquets, il peut faire glisser cette fenêtre de n positions et donc s'autoriser à envoyer un même nombre de

nouveaux paquets. Le récepteur fonctionne avec le même système de fenêtre qui indique chez lui les paquets qu'il est susceptible de recevoir et qu'il doit accepter. Le récepteur va passer les paquets de cette fenêtre à la couche application dans l'ordre logique et non pas spécialement dans l'ordre de réception. [L2]

On peut donc détecter ici une première entrave à l'utilisation de TCP dans le cadre d'un transfert d'informations sensibles au délai tel que MIDI. Imaginons que l'émetteur ait une fenêtre de 10 paquets et que le premier soit perdu. Le récepteur accepte le deuxième paquet, il transmet un accusé de réception négatif pour le premier paquet à l'émetteur, mais malgré le fait qu'il reçoive correctement les paquets suivants, il ne peut les transmettre à la couche application avant la retransmission du paquet perdu. Cela pose évidemment un problème car cette situation entraîne un décalage dans le temps non récupérable puisque le récepteur non plus ne peut avancer sa fenêtre et qu'il retarde donc les paquets que la couche application souhaite transmettre.

Soulevons un deuxième problème du même type ; pour rendre le transfert plus performant, l'implémentation TCP peut attendre d'avoir suffisamment de donnés avant d'envoyer un paquet sur le réseau.
Le troisième problème concerne le mécanisme mis en place dans les implémentations actuelles afin d'éviter la congestion du réseau. Il s'agit en fait d'adapter la taille de la fenêtre glissante en fonction des conditions réseaux. Grossièrement, si les conditions réseaux se dégradent, la taille de la fenêtre d'émission va être réduite d'un facteur deux. Cela va donc réduire le nombre de paquets que l'émetteur peut transmettre sans acquittement. Cette réduction de la taille de la fenêtre s'opère dès que l'émetteur constate qu'un paquet a été perdu. Si la transmission ne pose plus de problème, l'émetteur va très progressivement augmenter la taille de sa fenêtre. Si ce

mécanisme peut s'avérer efficace et n'aura pas de conséquence néfaste lors d'un transfert de fichier, il va par contre induire de la gigue sur nos transmissions MIDI. [A12]

Il est clair que nous aurions souhaité avoir un transport fiable, mais il nous est inacceptable de rajouter au temps de transmission un délai supplémentaire dû aux retransmissions éventuelles.

Le choix TCP ne s'avérant pas concluant, tournons-nous vers UDP afin de voir s'il permet de mieux répondre à nos attentes.

UDP (*User Datagram Protocol*) fournit un service en mode non connecté et sans reprise d'erreur. Il n'offre aucune fonctionnalité d'ordonnancement, ni de contrôle de flux. C'est donc un protocole de transport très simple, mais qui présente l'avantage d'un temps d'exécution très court, qui permet de tenir compte des contraintes de temps réel. Ce sont ces critères qui nous ferons opter pour un support UDP au niveau de la couche transport.

Choisir UDP ne signifie pas que nous nous désintéressons des mécanismes permettant d'avoir un flux ordonné et contrôlable, cela signifie simplement que n'ayant pas le moyen d'utiliser ces fonctionnalités au niveau de la couche transport, nous espérons pouvoir résoudre les problèmes au niveau des couches supérieures. Nous reportons donc simplement une partie du traitement à la couche qui va se trouver au-dessus et UDP, même s'il est incomplet pour l'action que nous souhaitons réaliser, présente l'avantage de ne pas induire de problèmes supplémentaires. Essayons donc maintenant de voir si le protocole RTP peut répondre à certaines de nos attentes.

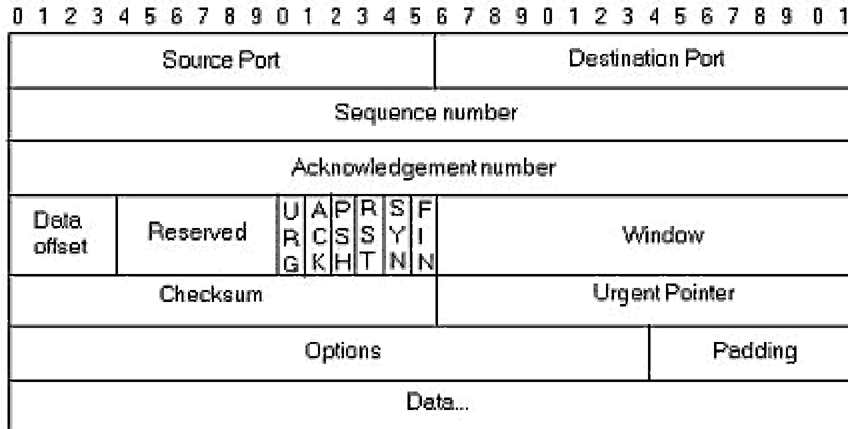
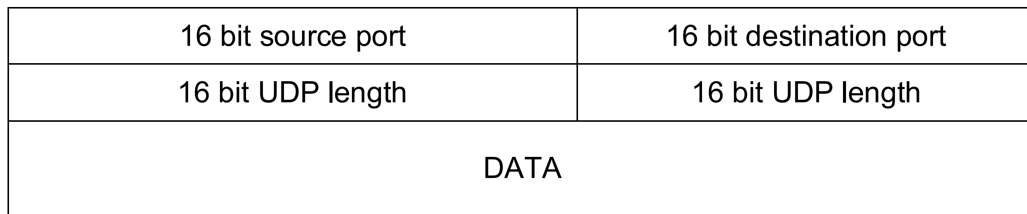


Figure 29 : Format de l’entête TCP



16 bit source port	16 bit destination port
16 bit UDP length	16 bit UDP length
DATA	

Figure 30 : Format de l'entête UDP

11 RTP / RTCP

11.1 RTP

TCP et UDP ne sont pas utilisables directement pour le transport des données à caractère temps réel à cause de leur délai et disponibilité imprédictible.

RTP (*Realtime Transport Protocol*) [A6] fournit un support pour les applications à caractère temps réel. Il permet la reconstruction sur le temps grâce à l'horodatage (*Timestamp*) qu'il inclut, la détection des pertes, car il numérote les données qu'il transmet, et enfin, la sécurité et l'identification du contenu transmis. Selon les nécessités, RTP pourra être utilisé avec ou sans le protocole de contrôle de flux RTCP (*Real Time Control Protocol*) qui lui est associé. RTP est indépendant de la couche transport sous-jacente de manière à ce qu'il puisse être utilisé au-dessus de protocoles de couche réseau tels que CLNP (*Connectionless Network Protocol*), IPX (*Internetwork Packet Exchange*) ou autres. La pile protocolaire Internet fournit deux protocoles au niveau de la couche transport TCP et UDP. TCP fournit un flux fiable entre 2 hôtes (*hosts*) et est orienté connexion, ce qui limite l'utilisation en *multicast*.

UDP fournit un service datagramme en mode non connecté, non fiable. Pour utiliser UDP comme protocole de transport temps réel, nous voulons lui ajouter certaines fonctionnalités. RTP est donc le protocole qui va venir compléter UDP avec les fonctionnalités dont nous avons besoin. Les applications font généralement évoluer RTP au-dessus d'UDP comme une part de la couche transport. RTP

dispose de fonctionnalités lui permettant de fonctionner aussi bien pour l'*unicast* que pour le *multicast*.

La session RTP

Pour initier une session RTP l'application va définir une paire d'adresses de destination (une adresse réseau et 2 ports, un port RTP / un port RTCP).

Dans une session multimédia, chaque type de média est transporté sur une session RTP différente et peut donc avoir son propre canal de contrôle RTCP qui indique la qualité de la réception. Cette séparation peut permettre au destinataire de sélectionner uniquement les canaux par lesquels il est intéressé, par exemple : abandonner la vidéo pour ne garder que le son.

Le paquet RTP

Les données utiles à l'intérieur des paquets RTP (voir Figure 31) sont précédées d'une entête et le message résultant est stocké dans un paquet UDP. L'entête RTP indique le type de codage utilisé (exemple : PCM). Les utilisateurs peuvent changer le codage durant la session selon la congestion ou d'autres paramètres (exemple : nouvel arrivant).

L'horodatage et le numéro de séquence de l'entête RTP permet la reconstruction du message après la transmission non fiable dans le réseau.

V	P	X	CC	M	Payload type	Sequence Number
Timestamp						
Synchronisation Source Identifier (SSRC)						
Contributing Source Identifier (CSRC)						
Profile dependent				Size		
Data						

Figure 31: Format du paquet RTP

Les douze premiers octets sont présents dans tous les paquets RTP, alors que la liste des identifiants CSRC (*contributing source*) est présente seulement quand le paquet est constitué par un mixeur. Le mixeur est un système intermédiaire qui reçoit des paquets RTP d'une ou plusieurs sources peut changer le format des données et combiner les paquets puis les transmettre comme un nouveau flux RTP.
Afin d'effectuer cette tâche le mixeur RTP pourra resynchroniser plusieurs sources.

Signification des différents champs :

V : version (2 bits) (nouvelle = 2)
P : un bit, mis à 1 il indique la présence d'octets de bourrage à la fin qui ne font pas partie de la charge utile du paquet. Le dernier octet du bourrage contient le nombre d'octets de bourrage qui doivent être ignorés.

X: Extension (1 bit). mis à 1 l'entête fixe est suivi exactement d'une extension de l'entête.
CC : CSRC count (4 bits) indique le nombre d'identifiant CSRC qui suivent l'entête fixe. Ce nombre est plus grand que un si la charge

utile du paquet RTP contient des données de sources différentes.

M: Marker (1 bit), il est défini par l’application, il est prévu pour signaler des événements particuliers dans le flux de données. Exemple : Données correspondant au passage à une nouvelle image dans un flux vidéo.

Payload type (7 bits). Il identifie la charge utile du paquet et permet à l’application d’agir en fonction.
Sequence number (16 bits). Il est incrémenté à chaque envoi de paquet RTP, il aide la source à réordonner les paquets reçus et à détecter les pertes. La valeur initiale est fixée aléatoirement.

Timestamp ou horodatage (32 bits). C’est l’instant d’échantillonnage du premier octet des données RTP. Peut être utilisé pour la synchronisation et le calcul de la gigue. Sa valeur initiale est fixée aléatoirement.
SSRC: 32 bits. C’est un nombre choisi aléatoirement qui permet de distinguer les différentes sources de synchronisation dans la même session RTP. Il indique si les données ont été combinées ou si la source de données est unique.

CSRC: 0 à 15 éléments, 32 bits chacun. C’est l’ensemble des sources contribuant au contenu du paquet. Le nombre d’identifiant est donné par le champ CC.

Les fonctionnalités de RTP

RTP fournit un service de bout en bout pour les données à caractère temps réel.
Les applications utilisent généralement RTP au-dessus d’UDP afin de pouvoir bénéficier des services de multiplexage et de checksum.

Mais beaucoup d'efforts ont été faits pour garder RTP indépendant de manière à ce qu'il puisse être utilisé sur d'autres protocoles.

RTP en soi ne fournit aucun mécanisme pour limiter les délais ou fournir d'autres garanties en terme de qualité de service, mais se fie aux services des couches inférieures. RTP suppose que le réseau sous jacent est fiable et qu'il fournit les paquets de manière ordonnée.

RTP est un cadre de travail protocolaire délibérément incomplet, la spécification plus complète du format de charge utile se fait selon les besoins spécifiques de l'application.
RTP fournit les fonctionnalités pour transporter du contenu temps réel tel que l'horodatage et permet de mettre en place des mécanismes de synchronisation entre différents flux. Par le fait que RTP/RTCP est responsable du contrôle de flux d'un type de média, il ne va pas automatiquement synchroniser plusieurs flux. Ceci doit donc se faire au niveau de la couche application. La mise en place de mécanisme de contrôle de flux et de congestion au niveau de l'application est facilitée par les rapports RTCP.

11.2 RTCP Real-Time Control Protocol

RTCP est le protocole de contrôle qui coopère avec RTP. Il fournit de l'information aux applications, sa première fonction est de fournir une indication de la qualité de distribution des messages RTP. Des expériences en *multicast* IP ont clairement démontré l'intérêt de RTCP dans le diagnostic des erreurs de distribution et dans le contrôle du flux.

Chaque paquet RTCP contient un rapport de l'émetteur et/ou du destinataire contenant des informations statistiques très utiles pour

l'application. Ces statistiques incluent le nombre de paquets envoyés, le nombre de paquets perdus, la gigue entre les arrivés. Ces informations vont permettre à l'émetteur de modifier sa politique de transmission afin de s'adapter à un environnement réseau fluctuant.

Pour l'identification RTCP transporte un identifiant au niveau transport par source RTP appelé *canonical name* (CNAME). Ce CNAME est utilisé pour garder trace des participants à une session RTP. Les récepteurs utilisent le CNAME pour associer plusieurs flux de données d'un participant donné à un set de sessions RTP concernées (par exemple pour synchroniser l'audio et la vidéo).

Pour éviter de surcharger les ressources réseaux et pour permettre une extension aisée du nombre de participants aux conférences RTP, le trafic de contrôle RTCP est limité à 5% maximum du trafic de toute la session. Cette limite se déduit du taux auquel les paquets RTCP sont transmis en fonction du nombre de participants à la conférence. Comme chaque participant envoie des paquets de contrôle à tout le monde, il est facile de calculer à quel rythme il faut envoyer les paquets RTCP.

Il faut distinguer les congestions transitoire et persistante. L'analyse du champ RTCP du rapport de l'émetteur indiquant la gigue entre les arrivées nous permet de mesurer l'évolution du service réseau et nous permet de constater la congestion avant qu'elle ne devienne persistante et ne génère des pertes de paquets.

Optionnellement RTCP peut être utilisé comme une méthode efficace pour envoyer une information minimale à tous les participants de la session. Par exemple, le nom du nouvel arrivant dans la session peut être communiqué à tout le monde. Les paquets de contrôle RTCP peuvent être transmis périodiquement d'un participant à une session

RTP vers tous les autres.

12 MIDI Versus VoIP

Si nous résumons les choix que nous avons effectués jusqu'ici, c'est à dire, le protocole IP au niveau de la couche réseau et le protocole UDP avec la combinaison RTP/RTCP il est évident que nous sommes sur une base totalement identique à celle établie pour le développement de la téléphonie sur IP. [L3]

Si le choix des protocoles utilisés pour véhiculer l'information d'un bout à l'autre de notre réseau est identique à la manière de procéder en VoIP [A13], le traitement à effectuer au niveau de l'émetteur et du récepteur sera propre à notre application. Nous avons effectivement pu constater que le transport du MIDI requiert certaines attentions particulières dès l'instant où le réseau sous-jacent n'est pas totalement fiable et/ou n'offre pas une distribution des paquets en respect d'un délai constant.

Nous avons déjà montré que la problématique ne se situait pas principalement au niveau du débit, mais que nous voulions surtout palier aux problèmes de gigue et de perte quand cela s'avère possible. Alors que les traitements réalisés par les systèmes d'extrémités en téléphonie IP consistent plutôt à compresser le débit délivré par l'échantillonnage de la voix.

De plus il faut bien se rendre compte que dans les applications basées sur voix, théoriquement du moins, chaque acteur d'une conférence prendra la parole à son tour, alors qu'en MIDI l'idée de base est de permettre une communication simultanée de plusieurs personnes. Outre le fait que le traitement en réception et en émission

est plus symétrique, il faut aussi indiquer que la réactivité qui s'impose doit être beaucoup plus satisfaisante afin de permettre aux multiples instrumentistes de synchroniser leur jeu.

Nous avons vu que RTP est un protocole volontairement incomplet, ainsi VoIP va le compléter de la manière qui correspond à ces besoins et nous ferons de même pour le MIDI.

13 H.323

Nous avons mis en évidence que la différence principale entre la façon de procéder en téléphonie IP et notre propre façon de transmettre MIDI se situe au niveau du traitement réalisé par les systèmes d'extrémités. Il est donc concrètement possible de matérialiser les traitements nécessaires au sein d'un *codec* similaire à ceux utilisés pour l'encodage de la voix ou de l'image. Dans cette situation, nous pouvons nous demander s'il n'est pas intéressant de conformer notre système de transmission MIDI à la recommandation H.323 [A14] afin de pouvoir bénéficier de la puissance des mécanismes impliqués de la même manière qu'en téléphonie IP. Analysons donc les fonctionnalités H.323 afin de juger si elles sont capables d'améliorer l'usage de notre propre système. [A15]

H.323 est la dénomination donnée à l'ensemble de standards permettant la mise en place d'un service multimédia sur un réseau à transmission par paquets. Il s'agit d'une recommandation édictée par l'Union internationale des télécommunications (UIT) [I2].

Le service multimédia consiste en l'échange de données brutes, de la voix ou de la vidéo, avec un intérêt particulier pour les transferts qui requièrent un service temps réel. La recommandation H.323 décrit les procédures pour les conférences vidéo et audio point à point et multipoint sur les réseaux à commutation de paquet. H.323 est donc constitué d'un ensemble de protocoles permettant la communication entre plusieurs objets.

La capacité offerte par H.323 de mettre en place, au niveau de la

couche application, la gestion de connexions, la facturation, le contrôle d'admission, la gestion de la bande passante, la négociation des paramètres de communication, la traduction d'adresse réseau en alias unique (E-mail, numéro de téléphone, etc.) ainsi que le succès dont il bénéficie font de lui un allié incontestable pour le déploiement de tout système de communication multimédia. [A16]

La traduction d'adresse réalisée par un serveur spécialisé, le *gatekeeper*, consiste en la mise en correspondance d'un alias (e-mail, numéro de téléphone) en adresse de transport. Ce rôle est comparable à celui d'un serveur DNS lorsqu'il traduit le nom de domaine Internet en adresse de transport IP. Le mécanisme décrit par H.323 est cependant plus souple puisque l'affectation d'un alias à une adresse IP est mise à jour dynamiquement dès que l'usager est rattaché au réseau, ce qui permet donc à celui-ci de bénéficier d'une mobilité et d'une accessibilité accrues.

Le contrôle d'admission est le mécanisme permettant de déterminer quelles sont les opérations qu'un terminal particulier est autorisé à effectuer.

Le MCU (*Multipoint Control Unit*) se charge de la négociation entre les différents terminaux lors d'une conférence avec plus de 2 participants. Les terminaux doivent s'enregistrer auprès du MCU afin de recevoir les informations nécessaires pour participer à la conférence : *codecs* à utiliser, type de conférence (audio-vidéo).

L'exploitation des fonctionnalités proposées par une infrastructure H.323 ne peut donc qu'étendre les variations d'utilisation d'un système de communication MIDI.

14 QoS

Nous avons jusqu'ici analysé les mécanismes à mettre en place au niveau des systèmes d'extrémités en faisant abstraction des systèmes intermédiaires : les nœuds du réseau. Nous n'avons à aucun moment considéré une stratégie capable d'améliorer les traitements que le réseau applique à nos flux d'informations. Une telle approche se fonde sur la volonté de garder les concepts étudiés indépendant d'une infrastructure matérielle particulière. En effet l'environnement du réseau peut être mixte et englober des ressources sous la gestion administrative d'entités différentes, comme c'est le cas sur Internet. [L4]

Pourtant, afin d'être suffisamment général, il est tout de même nécessaire d'examiner la manière dont le réseau pourra gérer le flux émis par notre application. Il est donc nécessaire de définir et de confronter les différents modèles de qualité de service que nous pourrions mettre en œuvre.

Cette approche est d'autant plus importante, que rien n'empêche l'utilisation du modèle de transmission MIDI dans le cadre d'un réseau privé qu'il est possible de configurer à souhait.

L'analyse de l'influence de la qualité de service permet aussi de comprendre quelles sont les conséquences du type de service qu'un fournisseur d'accès ou un opérateur offre à l'utilisateur sur un tel flux applicatif.

La qualité de service d'un réseau désigne sa capacité à transporter

dans de bonnes conditions les flux issus de différentes applications [L5]. Cette définition de la QoS réseau se traduit par les caractéristiques techniques suivantes :

La fiabilité : le service d'acheminement du réseau doit être fiable et disponible;

La bande passante : il doit proposer suffisamment de bande passante (débit) pour absorber les trafics générés par les utilisateurs ;

Le délai : il doit permettre aux trafics utilisateur qui le désirent un service d'acheminement rapide, l'appellation technique correspondant à latence ;

La régularité : il doit assurer aux trafics utilisateur qui le désirent un acheminement régulier du trafic, l'appellation technique étant gigue ;

Le taux d'erreurs : il doit assurer aux trafics utilisateur qui le désirent un service d'acheminement sans perte.

La mise en œuvre d'une solution globale de QoS impose de disposer de :

- mécanismes spécifiques de bout en bout du réseau (mécanismes horizontaux) permettant de signaler aux différents nœuds du réseau le comportement à adopter pour traiter tel ou tel flux issu d'une application.

- mécanismes horizontaux permettant d'offrir aux applications, par le biais d'une interface appropriée, la QoS requise en se fondant sur des mécanismes de plus bas niveau. Ainsi, les mécanismes de QoS mis en œuvre au sein des équipements réseaux (les routeurs) devront également se référer aux mécanismes de QoS des liens de communication utilisés (ATM, Ethernet, etc.). Il est important de remarquer que le QoS proposée par un niveau est dépendante d'une QoS de niveau inférieur.

14.1 Gestion de la QoS

Gestion en meilleur effort

La gestion en meilleur effort (*best effort*) correspond à l'absence de mécanismes de QoS sur le réseau. Les flux des applications expérimentent alors des conditions de traversée du réseau qui peuvent être aléatoires. C'est le cas actuellement du réseau Internet qui offre des conditions d'utilisation très variables. Il en va de même de la majorité des réseaux d'entreprise pour lesquels aucune politique de priorité des flux n'a été mise en œuvre sur les routeurs. Si la situation n'est pas gênante pour les applications de messagerie ou de transfert de fichier, elle devient critique pour les applications qui nécessitent le respect de délai ou de bande passante.

Les partisans de cette approche considèrent que l'évolution rapide des débits satisfait les besoins sans qu'il soit nécessaire de mettre en œuvre une gestion complexe des trafics.

Cette approche simpliste n'est valable que lorsque le coût d'implantation et de gestion de la QoS est trop important relativement à l'augmentation de qualité qu'elle engendre.

- Le surplus de bande passante coûte cher. Afin d'obtenir des performances acceptables il est nécessaire de surdimensionner le réseau pour pouvoir tenir compte des pics de trafics. Or, cela signifie qu'on investit pour une capacité qui ne sera, finalement, que rarement utilisée.

- Le surplus de bande passante ne privilégie pas le trafic prioritaire : les trafics sensibles au délai se trouveront toujours pénalisés par la politique FIFO pratiquée dans les nœuds du réseau.

Nous avons étudié des mécanismes capables de résoudre une part des problèmes posés par ce type de service pour peu qu'ils restent suffisamment circonscrits. Il serait toutefois plus intéressant de pouvoir bénéficier d'une qualité de service supérieure, car le transport du MIDI dans ce contexte ne sera efficace que si la charge du réseau est suffisamment faible.

Gestion du trafic au sein du réseau

La gestion du trafic au sein du réseau consiste à mettre en œuvre sur les équipements du réseau les mécanismes permettant de gérer prioritairement certains flux grâce à une analyse préalable (demande implicite) ou à une demande explicite de QoS (avant le transfert). Les flux moins prioritaires seront mis en file d'attente dans les nœuds du réseau et écoulés dès que possible, à moins qu'une saturation trop importante du réseau n'oblige à les détruire partiellement ou en totalité.
Cette gestion implique donc que les nœuds du réseau soient en mesure d'assurer les trois fonctionnalités suivantes :

- la classification qui permet de trier les trafics entrants pour les affecter à des files d'attente ;
- la mise en file d'attente : les files d'attente permettent le partage de la bande passante du lien de sortie ;
- l'ordonnancement qui permet de servir les files d'attente en fonction de leur priorité respective.

Afin de simplifier les équipements, certains modèles de QoS proposent d'effectuer la classification complexe des paquets à l'entrée du réseau pour leur affecter une marque. Les éléments centraux du réseau n'auront donc plus besoin d'opérer une classification

complexe des paquets, mais les ordonneront plus simplement en fonction de leur marque.

Gestion du trafic aux extrémités du réseau

Les gestionnaires de bande passante permettent de gérer le débit TCP (TCP *rate Control*), en contrôlant le débit des applications TCP en fonction des conditions de charge du réseau et de la priorité respective des applications.

La gestion de files d'attentes personnalisées (CBQ, *Custom Based Queuing*) consiste à affecter les flux à l'entrée du réseau à différentes files d'attentes, selon la nature de l'application et à partager la bande passante entre ces files selon la priorité affectée à chaque file.

14.2 Niveaux de service de la QoS réseau

L'usage semble distinguer trois niveaux principaux de service de QoS :

- Le service au mieux ou *best effort* : Il ne propose pas de QoS, mais un service primaire de connectivité entre deux points quelconques du réseau.
- Le service différencié : ce service propose un traitement préférentiel de certains types de trafic. Il s'agit d'une préférence relative et non d'une garantie absolue. En d'autres termes, les trafics prioritaires expérimenteront un meilleur délai d'acheminement que des trafics non prioritaires, mais il n'y aura pas de garantie sur une valeur du temps d'acheminement.

- Le service garanti : ce service propose des caractéristiques de QoS garanties grâce à la réservation de ressources sur le réseau.

Certaines entreprises et la majorité des opérateurs optent pour la mise en œuvre simultanée de ces trois niveaux de service. Dans ce cas l'utilisation du niveau de service sera fonction des applications et donnera lieu, le cas échéant, à la facturation correspondante. Les services garantis constituent alors les trafics prioritaires du réseau, les services différenciés le seront à un moindre niveau et, enfin les services au mieux seront les moins prioritaires.

Dans notre cas, le service différencié nous permet d'assurer que nos données ne vont pas être mises en file d'attente derrière du trafic de faible importance.

Le service garanti nous permet de réserver les ressources avant le transfert. Il faut toutefois préciser que si en téléphonie IP une estimation du besoin en bande passante peut-être définie avant l'utilisation du canal, il n'en va pas de même en MIDI étant donné que les données ne sont pas transmises à intervalles réguliers mais seulement lorsqu'une touche du clavier aura été activée. Nous pouvons faire une demande de réservation en nous basant sur le nombre de messages qu'une interface MIDI classique peut fournir et en recalculant le débit nécessaire sur base de la taille des paquets réellement transmis sur les réseaux, mais cela constitue un réel gaspillage de bande passante.

14.3 Les modèles liés au protocole IP

IntServ (RSVP)

Le modèle de gestion *IntServ* de l'IETF (Internet Engineering Task Force) [I3] définit, pour un système hôte, une demande de service spécifique à un réseau. Par service spécifique, on entend :

- Un délai de traversée du réseau,
- Une bande passante,
- Un seuil minimal de perte de paquet.

Ce modèle se fonde sur la réservation de ressources réseau par flux applicatif. Le protocole de réservation de ressources utilisé entre l'émetteur et le récepteur correspond au protocole RSVP (*Resource Reservation Protocol*). C'est pourquoi le modèle *IntServ* est souvent associé au protocole RSVP. Cette gestion définit un modèle de service comprenant à ce jour deux définitions :

- Charge contrôlée (ou CL, *Controlled Load*) : ce service propose aux applications une connexion de bout en bout de type *best effort*, mais dans des conditions de charge normale du réseau. Cela signifie donc qu'aucune garantie d'acheminement n'est offerte aux applications, mais que les trafics ne sont pas supposés expérimenter une congestion sur le réseau. En résumé, il s'agit d'un service supérieur au service *best effort* ;

- Service garanti (ou GS, *Guaranteed Service*) : il permet de proposer un acheminement du trafic avec bande passante et délai garantis.

Ce modèle est complexe à mettre en œuvre, car il suppose que chaque routeur du réseau mémorise un grand nombre d'informations (réservation de ressources pour chaque flux applicatif) et qu'il identifie la réservation enregistrée pour chaque paquet IP le traversant.

Diffserv

Le modèle *DiffServ* de l'IETF se fonde sur l'affectation d'un niveau de priorité à un ensemble de flux IP, regroupés en agrégat. L'identification du trafic IP en agrégat correspond au regroupement de flux possédant certaines caractéristiques communes : même préfixe d'adressage IP, même numéro de port, etc. En fonction de cette identification, une valeur de priorité est affectée dans l'en-tête du paquet IP. Les nœuds du réseau peuvent ainsi acheminer ce paquet en fonction de la valeur de l'information de priorité.

Le modèle *DiffServ* comprend donc la création de classes de trafic (en nombre limité) et la classification du trafic par rapport à ces classes placées à la périphérie du réseau. Ce modèle est plus évolutif que le précédent, car il n'est pas nécessaire de mémoriser les besoins individuels de chaque flux sur le réseau, mais seulement d'offrir un traitement différencié selon la classe d'appartenance des différents paquets IP.

Nous avions déjà opté précédemment pour le service différencié, *DiffServ* semble donc la stratégie de gestion de qualité de service la plus adaptée pour mettre en place de manière raisonnée un système de transmission MIDI basé sur les concepts développés dans ce mémoire.

15 Mesure de la qualité

Il est généralement intéressant de pouvoir effectuer une mesure de qualité d'un système se prétendant capable de reproduire une information quelconque. Dans le monde du son, deux approches sont utilisées pour évaluer la qualité d'un tel dispositif.

On peut obtenir des écouteurs humains une évaluation subjective et fiable de la qualité du son ou des signaux du discours, mais la procédure exige beaucoup de travail et de temps.

Progressivement des logiciels permettant d'évaluer et de mesurer de façon objective la qualité d'une séquence sonore traitée en regard d'une version originale non traitée de la même séquence sont mis au point.

Il faut tenir pour acquis que les deux versions sont simultanément disponibles dans les dossiers informatiques et qu'elles sont synchronisées en temps réel. Les séquences sonores sont traitées par un modèle de calcul de l'audition qui se libère des composantes auditives qui sont normalement imperceptibles par l'auditeur humain. La version traitée se compare à la version non traitée et la différence sert à prédire le taux moyen de qualité auquel devraient s'attendre les auditeurs (voir Figure 32). Ce type de logiciel permet ainsi de mettre à l'essai et comparer divers algorithmes *codec* de la parole et de l'audio ainsi que leurs mises en œuvre. L'Union internationale des télécommunications (UIT) décrit en détail une méthode normalisée pour jauger la qualité d'un signal sonore sur une grande largeur de bande (Recommandation BS-1387 de l'UIT). Le modèle psychoacoustique sur lequel se base ce modèle a pour acronyme

PEAQ (*Perceptual Evalutation of Audio Quality*).

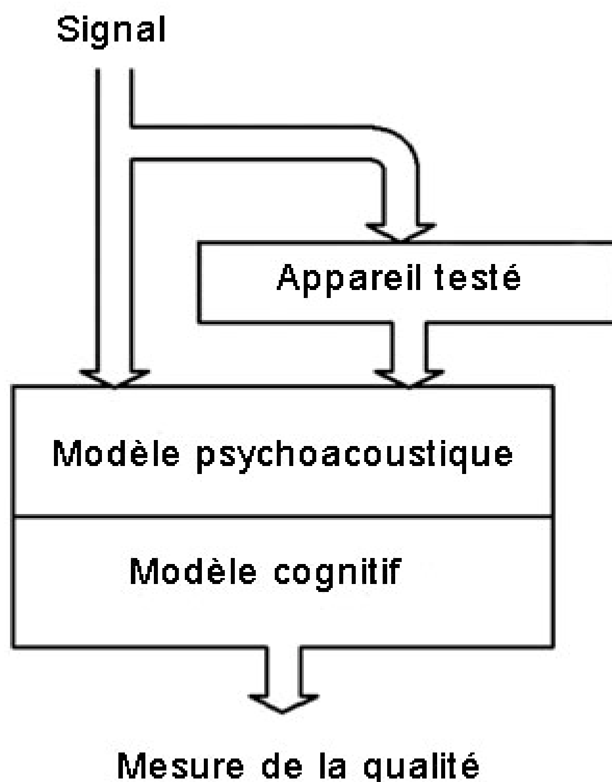


Figure 32 : Utilisation de modèles pour la mesure de qualité sonore

La fiabilité d'une telle mesure se base sur la précision du modèle psychoacoustique et cognitif mis en œuvre. Si une modélisation précise du comportement purement psychoacoustique permet d'utiliser avantageusement ce type de méthode pour mesurer la qualité sonore, il sera difficile de mesurer la qualité de nos transmissions MIDI avec l'aide d'un tel modèle. Effectivement, si nous nous sommes basés sur des principes psychoacoustiques pour détecter les dégradations perceptibles, la mesure de l'efficacité des décisions prises pour remédier à ces imperfections nécessite une modélisation plus complexe, reflétant les processus intellectuels mis en œuvre lors de l'écoute de la musique. La différence principale est que les *codecs audio* modifient essentiellement la qualité sonore, « l'orthographe de la musique », alors que les défauts en MIDI altèrent soit la structure mélodique et/ou harmonique du jeu musicale,

soit sa structure rythmique, « la grammaire de la musique ».

Il faut donc se résoudre à effectuer les mesures sur base de l'auditeur humain en comprenant que les observations glanées et l'analyse de ces résultats peuvent eux-mêmes contribuer à mieux comprendre les processus de compréhension de la musique. [L6]

16 Implémentation

Les divers concepts analysés ont été progressivement mis en œuvre au sein d'une implémentation pratique afin d'en vérifier aussi bien la validité que de permettre la quantification de leurs apports.

Le développement a suivi le cours de l'analyse théorique et a ainsi permis d'identifier clairement certains problèmes particuliers qui ne nous apparaissaient pas au premier abord. Ainsi l'application porte encore les marques du besoin d'avoir un contrôle sur de multiples paramètres et d'afficher un nombre de résultats totalement inutiles dans le cadre d'une utilisation conventionnelle. Les mêmes considérations sont à prendre en compte quant au défaut d'ergonomie évident.

Cependant l'application résultante s'avère fonctionnelle et met en évidence les considérations utiles pour le développement d'un produit fini.

Nous proposons donc ici de retracer l'évolution que l'application a connu, afin d'illustrer les considérations prises en compte, les résultats obtenus et les techniques mises en œuvre.

16.1 Mise en place des éléments de base

L'application a été réalisée dans l'environnement Microsoft Windows 2000 à l'aide de l'outil de développement Borland C++ Builder dans la version 4.

Acquisition et reproduction de données MIDI

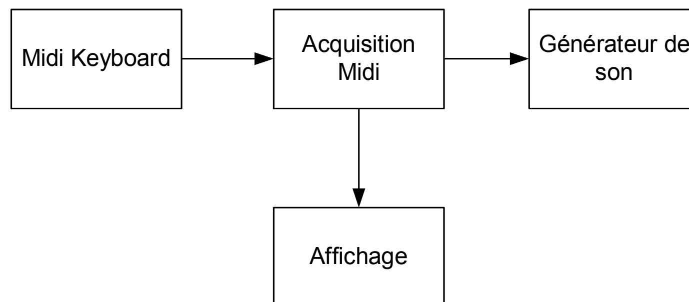


Figure 33 : Interaction avec le matériel MIDI

La première mouture de l'application (voir Figure 33) nous a permis de nous familiariser à l'interaction avec le matériel MIDI. Les fonctions de la librairie standard MMSystem ont été utilisées pour communiquer avec le matériel.

Le traitement des informations émises par l'instrument MIDI suppose de notre part la définition d'une fonction spécifique à cette tâche, dont l'adresse doit être fournie en paramètre lors de l'appel d'une fonction spécialisée du driver MIDI. Le driver fait appel automatiquement à cette fonction dès l'instant où il a pu remplir un buffer avec les données qu'il vient lui-même de réceptionner de l'instrument MIDI connecté à l'entrée de la carte son.

Nous avons donc, dans un premier temps, élaboré une fonction simpliste sans but plus ambitieux que l'affichage de l'input perçu ou la retransmission directe vers l'output de la carte son.

Nous avons ensuite analysé de plus près la structure des messages émis par un clavier MIDI et mesuré la quantité d'information transmise au cours du jeu sur ce même clavier.

Les cartes sons «grand public» actuelles peuvent se charger directement de l'interprétation des messages MIDI. Toutefois si un instrument est disponible et connecté à la sortie de la même carte, il peut être avantageux d'utiliser plutôt le générateur qu'il inclut. [L7]

En effet, nous avons pu constater dès les premiers tests, que les cartes son MIDI «grand public» présentent souvent un temps de réponse trop important et clairement perceptible. Si l'utilisation pour la restitution de fichiers MIDI ne s'en ressent pas, cette situation est beaucoup plus gênante pour des manipulations telles que celles que nous prévoyons. Il semble donc que l'acquisition d'une carte MIDI à ce dessein requière une attention particulière. Le temps de réponse d'un véritable instrument MIDI s'est quant à lui, sans surprise, avéré efficace pour le jeu interactif.

Simulation

L'observation du trafic MIDI a ensuite fait place au besoin de simuler la dégradation subie lors de la transmission sur les réseaux réels. La mise en place d'un module de simulation s'est donc avérée l'étape indispensable pour pouvoir aussi bien analyser le comportement d'un récepteur en cas de perte, que pour permettre d'établir des seuils de tolérance dans des conditions totalement contrôlables et indépendamment d'une architecture réseau particulière. (Figure 34)

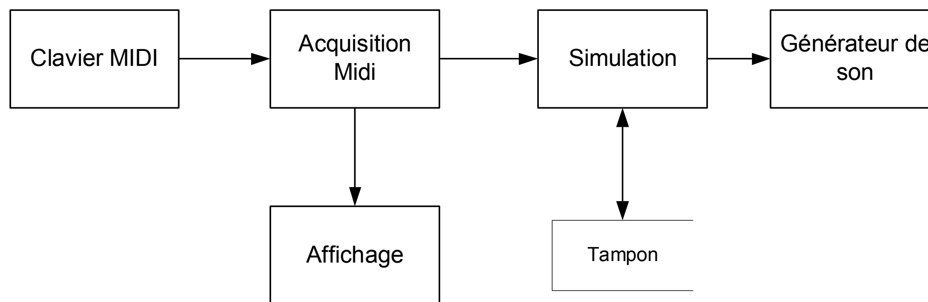


Figure 34 : Mise en place du module de simulation

Nous avons donc implémenté un module capable de simuler différentes situations sous une forme aisément paramétrable. Le module se charge de récupérer les donnés transmises par l'acquisition MIDI, de les manipuler et enfin de les transmettre vers le générateur de son.

Le module est développé pour simuler la perte d'information, la gigue et le délai.

Nous avons envisagé que selon l'indication du taux de perte fournie par l'utilisateur, le module puisse effectuer les modifications nécessaires sur le flux d'informations. Pour implémenter la perte, nous utilisons les fonctions de génération de nombres aléatoires. Nous spécifions un intervalle de 100 dans lequel le générateur va choisir un chiffre chaque fois qu'il est susceptible de transmettre une information. En estimant que la répartition des résultats du générateur aléatoire sur l'intervalle est uniforme, nous décidons que si le nombre choisi est plus petit que le taux de perte spécifié, le paquet est perdu et dans ce cas nous ne le retransmettons pas vers le générateur.

```
Pour chaque information à transmettre
   Si (( Choix aléatoire sur 100) <  % de perte indiqué * 100)
            Eliminer l'information
   Sinon
```

Transmettre l’information

Fsi

FPour

Pour pouvoir simuler un délai, il devenait nécessaire que le module puisse stocker les informations pour une période déterminée. Nous avons choisi de stocker les informations sous forme d’une liste chaînée dynamique. Le module insère de manière triée l’information reçue dans la liste en y ajoutant un horodatage permettant au module de simulation de n’extraire les données de la liste qu’après l’écoulement du temps déterminé par l’utilisateur. (voir Figure 35)

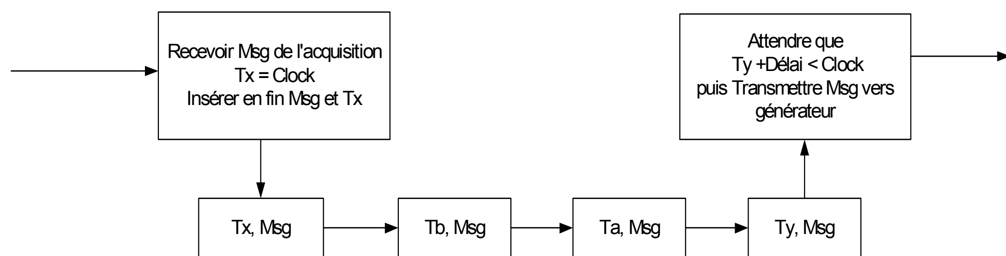


Figure 35 : Simulation du délai

La simulation de la gigue a été réalisée d’une manière analogue. Un temps de gigue est recalculé aléatoirement dans l’intervalle de gigue choisi par l’utilisateur (0 – gigue max.). Selon la nomenclature de la Figure 35, cette valeur calculée est ajoutée à Tx, et l’insertion n’est plus faite systématiquement en tête de liste mais de manière triée sur les valeurs (Tx + gigue aléatoire).

C’est à ce moment de l’implémentation que nous estimons que différentes tâches méritent d’être réparties en processus différents. Nous avons donc choisi d’implémenter les *threads* supportés par le

système d'exploitation Windows. Ce choix s'explique essentiellement par l'indépendance que nous souhaitons fournir aux différents modules plutôt que par la recherche du code le plus efficace.

L'adoption du *multithreading* nous impose de repenser la manière dont les modules s'échangent leurs informations. En fonction de nos objectifs, le choix de communiquer par liste chaînée a semblé le plus adéquat. Les mécanismes de verrouillage (*lockList*(), *unlockList*()) d'accès, disponibles dans les fonctions membres des *threadList,* semblent permettre une utilisation rationnelle de ce type de structure.

Expérimentation

La mise au point du simulateur nous a permis de procéder à différentes observations, selon la vitesse d'exécution et l'instrument utilisé:

- Détermination du temps de réponse « acceptable ». Voir page 54 : Table 4.
- Seuil de perception de la dégradation par l'introduction de gigue. Voir page 94 Table 5
- Impact des pertes éventuelles de messages MIDI.

Ces observations ont été réalisées sur base de morceaux préenregistrés dans le synthétiseur (le clavier utilisé permet de restituer des séquences MIDI préenregistrées) ou du jeu réel sur le clavier.

Le module simulation, malgré sa simplicité a gardé une importance prépondérante dans l'ensemble du travail.

Gigue	BPM = 60	BPM = 90	BPM = 120	BPM = 160
0 ms	Non	Non	Non	Non
0 – 50 ms	Non	Non	Non	Non
0 – 100 ms	Non	Oui	Oui	Fort
0 – 150 ms	Léger	Oui	Fort	Fort

Table 5 : Perception de la gigue en fonction du tempo

Générateur de notes

Il nous a cependant semblé utile, à un certain moment, d'avoir des séquences répondant à des critères temporels plus précis. Nous avons donc décidé d'incorporer un générateur MIDI aléatoire à notre application. Le principe de fonctionnement est relativement primaire, mais il nous a tous de même permis de procéder à l'observation d'un signal dont nous connaissions précisément les caractéristiques. Nous avons, par exemple, réalisé quelques tests sur des séquences de notes apparaissant à intervalles réguliers, auxquelles nous avons appliqué une gigue de x%.

Il est à noter que nous avons tenu compte d'une part, que les différentes observations réalisées par un échantillon très restreint doivent être interprétées avec prudence et d'autre part, que les dégradations de séquences régulières ou aléatoires ne sont certainement pas perçues de la même manière que les dégradations subies sur des séquences musicalement plus cohérentes.

Emetteur Récepteur RTP

La transmission réelle par le protocole RTP fut l'étape suivante de l'implémentation. Nous avons utilisé dans ce cadre la librairie JRTPLIP [A18] développée par Jori Liesenborgs [A18] en concordance avec la RFC 1889.

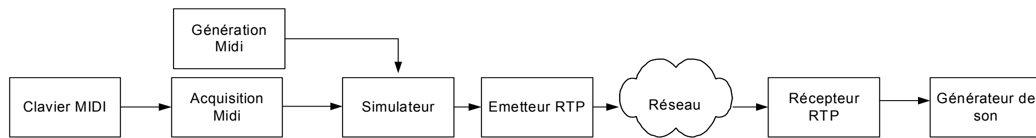


Figure 36 : Intégration des fonctionnalités RTP

L'adjonction des modules RTP a donc permis à notre système de communiquer avec l'extérieur. La mise en œuvre de la communication n'a pas posé de problème particulier. Nous n'utilisions pas encore de correction d'erreur et procédions donc simplement à l'envoi et la réception de paquet.

Suite à l'introduction des modules RTP, le rôle du simulateur s'est considérablement réduit et son adaptation a donc été nécessaire.

Il est à noter que RTP nous autorise à transmettre vers notre propre adresse IP, mais en utilisant un numéro de port différent, et donc, d'exécuter des communications et les divers tests sur une machine unique.

Le relais UDP

Il est apparu que l'insertion du module RTP ne nous permettait plus d'effectuer les simulations, car RTP se charge lui-même de la gestion de l'horodatage et il n'est donc plus intéressant de manipuler les informations à l'intérieur de l'application.

Outre l'horodatage, nous avons souhaité pouvoir prendre en compte la gestion des rapports de réception que RTP implémente afin d'informer l'émetteur de la qualité de service offerte par le réseau. [A6]

Or, le simulateur tel que nous l'avions mis en place n'est pas

transparent car RTP nous demande de lui indiquer l'intervalle depuis la dernière transmission et calcule, à partir de ces indications, différentes propriétés de la transmission : la gigue, le délai, le taux de perte.

Le simulateur ne pouvant donc plus fonctionner en amont de l'émission RTP, nous avons décidé de reprendre ses fonctionnalités dans un module indépendant et de le faire travailler au niveau de la couche UDP. Notre relais UDP écoute donc le port sur lequel l'application émet les paquets par RTP, et quand l'information est émise, il récupère le paquet UDP dans lequel les informations sont encapsulées. L'action du relais UDP au niveau de la couche inférieure à RTP le rend transparent du point de vue de l'application.

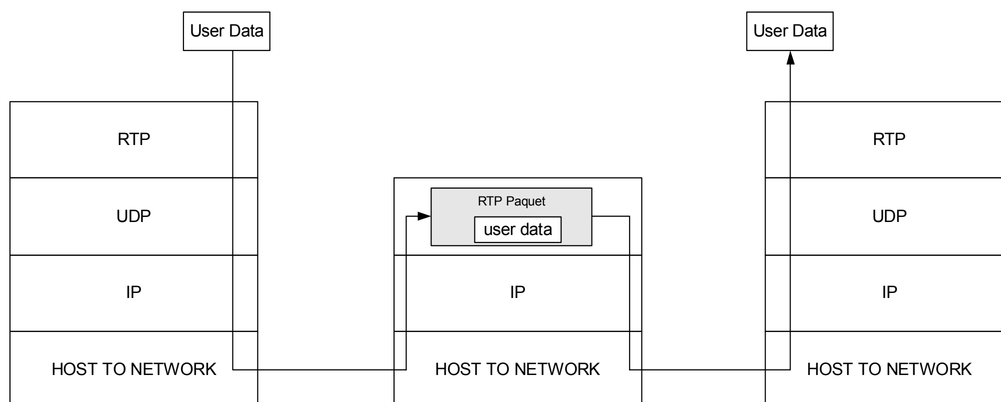


Figure 37 : Localisation de l'action du relais UDP

Comme le montre la Figure 37, le relais UDP récupère le paquet UDP complet, il décide ensuite s'il le détruit ou s'il se contente de le retarder, mais dans ce cas les informations contenues dans le paquet-même restent intactes. Les paquets ainsi traités sont finalement transmis à la couche RTP de l'autre instance de notre application, qui déduira, du retard qu'elle constate sur les données, que le service réseau est dégradé. Nous pouvons ainsi observer

clairement les comportements qui se seraient mis en place dans une situation comparable sur un réseau réel.

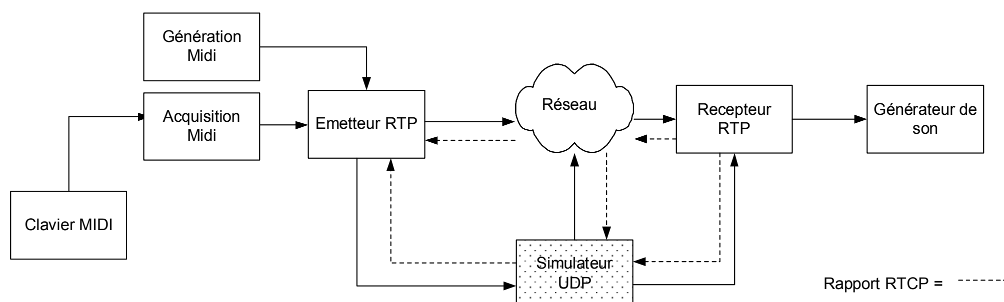


Figure 38 : Intégration du relais UDP

La Figure 38 nous montre l'intégration du simulateur-relais UDP à notre système de communication.

Nous avons mis en place les composantes nécessaires pour évaluer les concepts abordés au cours de notre étude, pour tester dans des conditions réseaux particulières (Simulateur UDP) tant sur une machine unique, que sur un réseau réel. Nous avons donc décidé à ce moment de procéder à l'intégration des mécanismes de correction.

16.2 Mise en place de mécanismes de correction

Nous avons matérialisé la correction au niveau de deux nouveaux modules : « préparation de l'envoi » et « préparation de la réception »

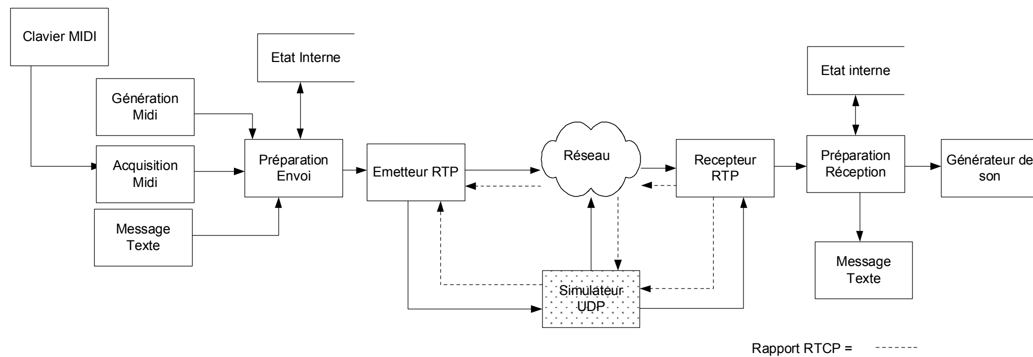


Figure 39 : Intégration des modules de gestion d'envoi et de réception

Avant de parler du fonctionnement de ces modules, signalons la présence du module Message Texte (Chat) qui permet de manière pratique de communiquer pendant les phases de tests à distance.

Correction basique

La première correction est mise en place au niveau du récepteur, programmé pour qu'il stocke l'état actuel du générateur. C'est à dire qu'il garde une image de son état en mémorisant la dernière note envoyée au générateur pour chaque touche. La structure de données utilisée est une simple table dont les indices permettent d'accéder directement aux informations enregistrées pour chaque note. La table doit donc contenir 128 entrées.
Pour chaque entrée les champs suivants sont disponibles :

- Numéro de séquence
- Horodatage
- Statut : actif ou passif

Nous utilisons donc cette table avant de transmettre les messages aux générateurs. Le test le plus simple consiste à vérifier que l'information reçue est plus récente que la dernière. Ce simple test nous permet de réduire les problèmes dus à la gigue.

Transmission de l'état de l'émetteur

La perte d'informations est le second problème pour lequel nous souhaitons mettre en œuvre les solutions proposées. Nous décidons donc d'inclure chez l'émetteur lui-même un état du système basé sur les informations émises précédemment. Pour remédier aux problèmes de pertes, nous allons adjoindre à chaque message une table binaire comprenant l'état du système selon la représentation binaire notes actives et passives (True et False). (Voir la partie 8 : Vérification de l'état du système).

Lorsqu'une information nécessite une émission, nous mettons à jour la table présente chez l'émetteur, insérons le message actuel et recopions la table sous forme binaire à l'intérieur du message.

Le récepteur va ainsi chaque fois avoir la possibilité de vérifier si certaines infos ne correspondent pas à l'état supposé par comparaison de la table reçue et celle qu'il conserve. En effet, le numéro de message fourni par RTP quand il reçoit la table, lui permet de voir si l'information est plus récente que celle dans sa propre table.

Un problème se pose toutefois, étant donné que nous avions prévu de ne plus jouer les notes qui arrivent trop tardivement. Or l'information binaire n'indique pas l'instant d'apparition de l'action. Comme il apparaissait intéressant d'exploiter cette possibilité, nous avons ajouté une seconde table de 128 bits indiquant simplement si les notes sont récentes. L'émetteur sait ainsi sans ambiguïté à quel moment la note a été générée puisqu'il conserve les indications nécessaires dans son état interne. Il calcule donc selon une valeur déterminée si la note est récente. Selon sa politique, l'émetteur va donc décider de ne pas procéder à l'activation de notes qu'il découvre

tardivement. Par contre, rien ne devra empêcher le récepteur de désactiver les *Note Off* qui ne sont pas récentes, si cela n'a pas été fait précédemment.

Intervalle de transmission de l'état du système

Nous avions estimé que l'envoi répétitif de paquet n'est pas intéressant (voir partie 6 : Intervalle de transmission), mais nous souhaitons tout de même effectuer des retransmissions entre les envois. Nous avons donc prévu de simplement retransmettre l'état binaire du système à intervalle croissant après l'envoi d'un message, en définissant un plafond de 500ms.

Message - 5 ms - 10 ms - 20ms - 40 ms - 80 ms - 160 ms - 320 ms – 500 ms –500 ms –500 ms …

Ce mécanisme permet de réduire le débit en gardant une bonne vitesse de retransmission si le taux d'erreur est relativement faible, et de remettre le système en état relativement rapidement si une perte en rafale à eu lieu. Les envois à intervalle progressif montre clairement un délai de réaction rapide et une correction généralement efficace.

Nous avions envisagé de transmettre les notes d'un accord au sein du même paquet, l'implémentation nous mène ici à considérer le contraire. En effet, le jeu est intensifié au moment où les accords sont joués mais comme un message est envoyé pour chaque note de l'accord et qu'il comprend une vue complète du système, malgré les pertes, les accords se transmettent assez bien (même sans la retransmission à intervalle progressif).

Temporisation

La dernière technique consistait à agir au niveau du récepteur pour pallier le délai induit (voir partie 9 : Mécanisme d'anticipation), l'idée étant de mettre à contribution la capacité d'anticipation de l'instrumentiste. Nous avons donc implémenté un *buffer* qui est respecté par l'émetteur et le récepteur. La taille du *buffer* est transmise au récepteur et s'il active le mode de correction, les notes seront émises chez lui au même instant, à condition bien sûr que le délai du réseau soit égal ou inférieur à cette valeur.

Nous avions pu montrer que la vitesse du jeu influençait fortement le temps de réponse acceptable du clavier. Nous avons donc ajouté un mode automatique qui adapte la taille du *buffer* à la vitesse du jeu. En reprenant (voir Table 4) des données que nous avons déjà pu analyser nous constatons que le temps de réponse acceptable varie linéairement avec la vitesse du jeu. Nous avons donc extrait une formule approximative : Temps de réponse = -6/14 * vitesse du jeu + 125 (voir Figure 40)

Temps de réponse	BPM = 60	BPM = 90	BPM =130	BPM =160	BPM = 200
0 ms	Non	Non	Non	Non	Non
20 ms	Non	Non	Non	Non	Léger
40 ms	Non	Non	Léger	Léger	Fort
60 ms	Léger	Léger	Oui	Fort	Fort
80 ms	Léger	Oui	Fort	Fort	Fort
100 ms	Oui	Fort	Fort	Fort	Fort
150ms	Fort	Fort	Fort	Fort	Fort
200 ms	Fort	Fort	Fort	Fort	Fort

Table 6 : Niveau de perception en fonction du temps de réponse et du tempo

Pour pouvoir appliquer cette formule, nous avons donc cherché à extraire la vitesse du jeu en temps réel : Pour cela nous avons programmé une fonction calculant le nombre moyen d'événements par minute en se basant sur un passé de x secondes (x devant être suffisant pour éviter qu'un léger silence dans le jeu musical ne soit

interprété comme une vitesse nulle). Cependant, la fonction peut être appelée plus fréquemment, par exemple, toutes les x/5 secondes pour un passé de x secondes, de manière telle que les accélérations soient assez rapidement prises en compte.

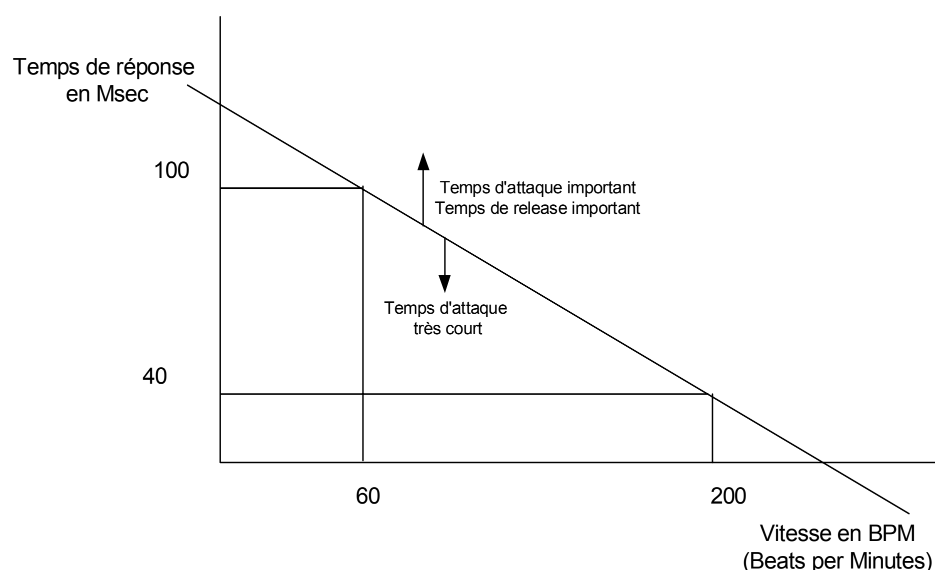


Figure 40 : Linéarisation du rapport Temps de réponse – Vitesse de jeu

Cette dernière méthode s'avère très efficace, car d'une part, elle diminue l'influence du délai de transmission et de l'autre, elle permet au récepteur de diminuer les problèmes de gigue et s'adapte finalement à la vitesse de jeu, de manière à offrir un intervalle de correction important pour le jeu lent et un temps de réponse satisfaisant pour le jeu rapide. Nous pouvons borner les valeurs que la taille du buffer peut adopter. Une variation entre 30ms et 150ms semble pouvoir représenter un bon compromis.

Une dernière nuance de ce calcul a encore été apportée par la prise en compte des caractéristiques ADSR du timbre utilisé (voir partie 7 : Pré-Analyse du comportement MIDI). Un temps d'*attack* et un *release* très longs rendent la détection du temps de réponse moins

précise, par contre, un temps d'*attack* très court va augmenter la perception des retards. (voir partie 4 : Perception musicale).

Ces dernières modifications terminent la description de notre implémentation MIDI. (Figure 41)

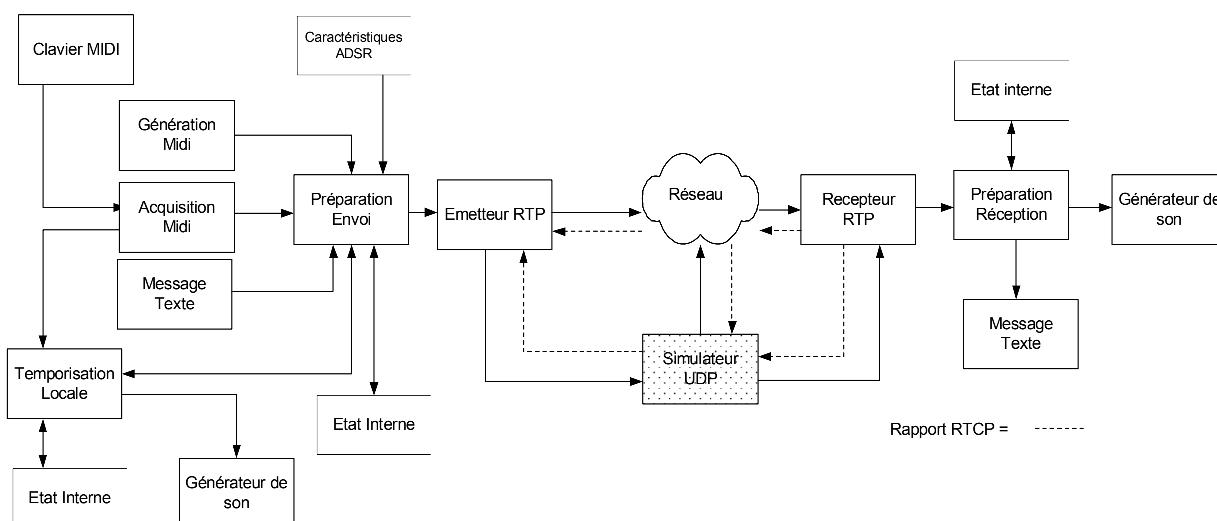


Figure 41 : Schéma complet de l'application NetMidi

17 Conclusion

La double approche utilisée - l'analyse globale de la problématique du transport MIDI ainsi que la définition de mécanismes appropriés pour l'amélioration du service offert - n'a été adoptée que dans la volonté, d'une part, d'indiquer les acquis réutilisables issus de domaines aussi variés que ceux de la psychoacoustique ou de la téléphonie IP et d'autre part, de construire originalement les pièces manquantes pour assembler ces différents éléments en un système fonctionnel.

De l'analyse, il ressort que le protocole MIDI n'est pas le candidat idéal pour un transport sur un réseau non fiable, étant donné que sa fonction primaire est de véhiculer les actions de l'instrumentiste et non de décrire la musique générée elle-même. Toutefois, nous avons pu mettre en évidence qu'une prise en compte d'un éventail plus large de paramètres et une mise en place de la Qualité de Service sur les réseaux actuels peuvent offrir un confort d'utilisation appréciable.

Si comme nous l'avons évoqué, l'augmentation de la bande passante disponible ne représente pas une solution miracle, elle aura pour le moins de bonnes chances d'autoriser ce type de communication si elle se trouve associée à des mécanismes évolués de contrôle de flux et de gestion d'erreurs comparables à ceux que nous avons évalués. Dans l'attente d'une pareille évolution pratique des réseaux, il est toutefois possible d'exploiter ponctuellement les lignes Internet ou de mettre en place une telle communication sur une infrastructure privée.

Bibliographie

[LX] Livres

[L1] Stevens R. W. (1994), *TCP/IP Illustrated, Volume I: The protocols*, Addison-Wesley.

[L2] Tanenbaum, A. S. (1996), *Computer Networks,* 3rd edition, Prentice Hall PTR, New Jersey.

[L3] Pujolle, G. (2000), *Les réseaux*, 3ème éd., Editions Eyrolles.

[L4] Ferguson P. & G. Huston (1998), Quality of Service: Delivering QoS on the Internet and in Corporate Networks, Wiley Computer Publishing.

[L5] Mélin, J.-L. (2001), *Qualité de service sur IP,* Editions Eyrolles, Paris.
[L6]McAdams, S. & I. Deliège (1988), *La musique et les sciences cognitives* (actes du Symposium sur la musique et les sciences cognitives, 14-18 mars 1988, Centre national d'art et de la culture Georges Pompidou, Paris), Edition Mardaga.

[L7] Stolz A. (1994), *SoundBlaster*, Editions Micro Application, Collection Grand Livre.

[L8] Zwicker, E. & R. Feldtkeller (1981), *Psychoacoustique : l'oreille, récepteur d'information*, Editions Masson.

[AX] Articles, documents techniques et de travail

[A1] Li, L., A. Karmouch & N.D. Georganas (1994), "Multimedia Teleorchestra with Independent Sources: Part 1: Temporal Modelling of Collaborative Multimedia Scenarios", *ACM/Springer on Multimedia Systems*, vol.1, n.4.

[A2] Young, J.P. & I. Fujinaga (1999), "Piano Master Classes via the Internet", Peabody Conservatory of Music, Johns Hopkins University.

[A3] Messick, P. (1997) "Maximum MIDI, Music Applications in C++" <http://www.maxmidi.com>

[A4] Rasch, R. A. (1978), “The perception of simultaneous notes such as in polyphonic music“, *Acustica,* 40, p.21-33.

[A5] Sorkin, R. D., G. J. Boggs & S. L. Brady (1982), “Discrimination of temporal jitter in patterned sequences of tones”, *Journal of Experimental Psychology: Human Perception and Performance,* 8, p.46-57.

[A6] Schulzrinne, H., S. Casner, R. Frederick & V. Jacobson (2001), “RTP: A Transport Protocol for Real-Time Applications”, *IETF* Internet Draft.

[A7] Casner, S. & V. Jacobson (1999), “Compressing IP/UDP/RTP Headers for Low-Speed Serial Links”, Network Working Group Memo, RFC2508.

[A8] Lazzaro, J. & J. Wawrzynek (2002), “MWPP: A resilient MIDI RTP packetization for network musical performance”, *IETF* Internet Draft.

[A9] YAMAHA Corporation (1996), “Proposal for Audio and Music Protocol”, Draft Version 0.32
<http://www.yamaha.co.jp/tech/1394mLAN/mlan.html>

[A10] YAMAHA Corporation (1997), “Specification for Audio and Music Data Transmission”, Working Draft Version 0.90f3.
<http://www.yamaha.co.jp/tech/1394mLAN/mlan.html>

[A11] Johansson, P. (1999), “IPv4 over IEEE 1394 “, Network Working Group Memo, RFC2734.

[A12] Stevens, W. (1997), “TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms”, Network Working Group Memo, RFC2001.

[A13] Rakesh, A. (1999), “Voice over IP : Protocols and Standards”, Ohio State University, Student Reports on Recent Advances in Networking.

[A14] Union internationale des télécommunications (UIT), “H.323 - Systèmes de communication multimédia en mode paquet“, (Recommandations UIT-T).

[A15] ElGebaly, H. & J. Toga (1998), “Demystifying Multimedia Conferencing Over the Internet Using the H.323 Set of Standards”, *Intel Technology Journal*.

[A16] Karim, A. (1999), “H.323 and Associated Protocols”, Ohio State University, Student Reports on Recent Advances in Networking.

[A17] Baker, F., J. Krawczyk & A. Sastry (1997), “Integrated Services Management Information Base using SMIv2”, Network Working Group Memo, RFC2213.

[A18] Liesenborgs, J. (2002), “JRTPLIB v. 2.6”.
<http://lumumba.luc.ac.be/jori/jrtplib/jrtplib.html>

[A19] Lazzaro, J. & J. Wawrzynek (2001), “A Case For Network Musical Performance”, Paper presented at the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video, Port Jefferson, NY.
[A20] Xu, A., W. Woszczyk, Z. Settel, B. Pennycook, R. Rowe, P. Galanter, J. Bary, G. Martin, J. Corey, and J. R. Cooperstock (2000), “Real-Time Streaming of Multichannel Audio Data over Internet”, *Journal of the Audio Engineering Society*, vol. 48, n.7/8.

[A21] Hall, B. “Beej” (2001), “Beej’s Guide to Network Programming. Using Internet Sockets”.

[A22] El Gebaly, H. (1998), “Characterization of Multimedia Streams of an H.323 Terminal”, *Intel Technology Journal*.

[A23] Foo, S. & S. C. Hui (1998), “A Framework for Evaluating Internet Telephony Systems”, *Internet Research*, 8(1), p.14-25.

[A24] Sullivan Jr., D.S., S. Moore & I. Fujinaga (1998), “Realtime Software Synthesis for Psychoacoustic Experiments”, Peabody Conservatory of Music, Johns Hopkins University.

[A25] Lucent Technologies, Schulzrinne, H., J. Lennox, D. Rubenstein & J. Rosenberg (1998), “RTP Library API Specification”.

[A26] Lucent Technologies (1997), “RTP Library Example User Code”.

[A27] ten Hoopen, G., E. van Buuringen, S. van den Berg & J. Memelink (1998), “Conceptions and Misconceptions of Discriminating Isochronous and Anisochronous Sound Sequences: A Crash Course in Classic Psychophysics”, 7th Workshop on Rhythm Perception and Production, Wassenaar, The Netherlands.

[A28] Repp, B. (1998), “Compensation for Subliminal Timing Perturbations in Perceptual-Motor Synchronization”, 7th Workshop on Rhythm Perception and Production, Wassenaar, The Netherlands.

[A29] Wessel, D. & M. Wright (2001), “Problems and Prospects for Intimate Musical Control of Computers”, CHI'01 Workshop New Interfaces for Musical Expression (NIME'01), Seattle, USA.

[A30] Heckroth, J. (1994), “A Tutorial on MIDI and Wavetable Music Synthesis”. <http://kingfisher.cms.shu.ac.uk/midi/main_p.htm>

[A31] Young, H. (1999), “Network MIDI -Transmitting Music via MIDI over a Local Area Network”, Indiana University Southeast, Computer Science Department.

[A32] Kerr, P. (2001), “DMIDI - Distributed MIDI Protocol”, Release 1.0.

< http://plus24.com/dmidi/ >
[A33] Lazzaro, J. & J. Wawrzynek (2002), "The MIDI Wire Protocol Packetization (MWPP)", *IETF* Internet Draft.

[A34] The Association of Musical Electronics Industry (Tokyo, Japan) & the MIDI Manufacturers Association (Los Angeles, USA) (2000), *MIDI Media Adaptation Layer for IEEE-1394 MMA/AMEI RP-027*, Version 1.0.

[A35] The International Engineering Consortium (2001), "Accelerating the Deployment of VoIP and VoATM", IEC Web ProForum Tutorial.

[IX] Sites Internet

[I1] The MIDI Manufacturers Association, Los Angeles, USA <www.midi.org>

[I2] ITU Telecommunication Standards
<http://www.itu.int/publications/main_publ/itut-fr.html>

[I3] The Internet Engineering Task Force <www.ietf.org>

[I4] Internet FAQ Consortium <http://www.faqs.org/>

[I5] The Association of Musical Electronics Industry (Tokyo, Japan)
<www.amei.or.jp>

[I6] The International Engineering Consortium www.iec.org

[I7] Yamaha mLAN <http://www.yamahasynth.com/pro/mlan/index.html>

Annexes

I Code source de l'application

MidiRead.h

```
//---------------------------------------------------------------------------

#ifndef MidiReadH
#define MidiReadH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <mmsystem.h>
#include "FormCtrl.h"
//---------------------------------------------------------------------------
class ThreadMidiRead : public TThread
{
private:
        unsigned char SysXBuffer[256];

//flag indiquant si on recoit un msg sysx
        unsigned char SysXFlag ;
            HMIDIIN                         handle;
            MIDIHDR                         midiHdr;
            unsigned long   err;

protected:
        void __fastcall Execute();
public:
        __fastcall ThreadMidiRead(bool CreateSuspended);
};
//---------------------------------------------------------------------------
#endif
```

MidiRead.cpp

```
#include <vcl.h>
#pragma hdrstop

#include <dos.h>

#include "rtpsession.h"
#include "rtppacket.h"
#include "ListMidi.h"
#include "MidiRtp.h"
#include "FormCtrl.h"
#include "FonctionUtile.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------
//   Important: Methods and properties of objects in VCL can only be
//   used in a method called using Synchronize, for example:
//
//      Synchronize(UpdateCaption);
//
//   where UpdateCaption could look like:
//
//      void __fastcall Unit1::UpdateCaption()
```

```
//      {
//        Form1->Caption = "Updated in a thread";
//      }
//---------------------------------------------------------------------------

__fastcall MidiRTP::MidiRTP(bool CreateSuspended)
        : TThread(CreateSuspended)
{
for (int i = 0 ; i < 128 ; i++)  // on met à jour la liste des temps
        {
              StatusSend[i].TimeStamp = 0;
              StatusSend[i].Status  = false;
        }

}
//---------------------------------------------------------------------------
void __fastcall MidiRTP::Execute()
{
bool errorRTP = false;

if(WSAStartup(0x101, &wsaData))
     {
        FormCtl->EditStatutConnect->Text =  "Unable to initialize WinSock library.\n";
        exit;
     }

        sess.Destroy();

        portbase = StrToInt(FormCtl->EditPortLocal->Text);

        destip = inet_addr(FormCtl->EditDestIp->Text.c_str());
             if (destip == INADDR_NONE)
             {
                           FormCtl->EditStatutConnect->Text = "Bad IP address specified";
                exit;

             }

             // The inet_addr function returns a value in network byte order, but
             // we need the IP address in host byte order, so we use a call to
             // ntohl
             destip = ntohl(destip);


             destport = StrToInt(FormCtl->EditPortDistant->Text); //PORT DE DESTINATION

     hostent *p;
     int i;
     char s[128];
     char *p2;
     //Get the computer name
     gethostname(s, 128);
     p = gethostbyname(s);
     FormCtl->MemoIpAddr->Lines->Add(p->h_name);


//    Get the IpAddresses
```

```
i = 0;
while (p->h_addr_list[i] != NULL)
   {
   p2 = inet_ntoa(*((in_addr *)p->h_addr_list[i]));
   FormCtl->MemoIpAddr->Lines->Add(p2);
   i++;
   }

         /*
            Now, we'll create a RTP session, set the destination, send some
            packets and poll for incoming data.
         */

         status = sess.Create(portbase);
      if (status < 0) errorRTP = true;
      sess.SetTimestampUnit(1.0/1000.0);



         status = sess.AddDestination(destip,destport);
      if (status < 0) errorRTP = true;
      if (!errorRTP)
        {
        SendReceive(); // on lit on ecrit alternativement.
        FormCtl->EditStatutConnect->Text = "RTP Initialisé";
        }
      else
       FormCtl->EditStatutConnect->Text = "Erreur RTP";
}
//---------------------------------------------------------------------------
void __fastcall MidiRTP::SendReceive()

{

char buffer[32];
char *DataPtr;

struct time Now;
unsigned long  TimeInMsec = 0 ;
unsigned long  LastSendTimeInMsec = 0 ;

__int8 temp;
__int8 NoteOn[36];
int Pos;
int Delay = 0;

bool NoteOnRecStatus[256]; // 4 premier bytes sont le message ,
//les 32 bytes suivants représentes les statuts : actifs et récent
bool NoteOnRecRecent[256];
bool BoolTemp;
bool Continuer;

DWORD msg;
AnsiString ChatMsg;

RTPSourceData *CurrSource;
```

```
PtrPaquet Paquet, PaquetBidon;
timeval TimingDelay;

NbrPaquetRec = 0;
NbrPaquetSend =0;


   while (FormCtl->SendingRTP)//
   {
   Sleep(2);

   status = sess.PollData();

         // check incoming packets

   if (sess.GotoFirstSourceWithData())
   {

    CurrSource = sess.GetCurrentSourceInfo();
    CurrSource->SetTimestampUnit(1.0/1000.0); // les incrementations de timestamp se font
par 1ms

         do
         {
               RTPPacket *pack;

               while ((pack = sess.GetNextPacket()) != NULL)
               {
                 // You can examine the data here
                 NbrPaquetRec++ ;
                 msg =  (*pack->GetPayload());

                 DataPtr = pack->GetPayload();


                 FormCtl->EditJitter->Text =   IntToStr(CurrSource->INF_GetJitter());
                 TimingDelay = (CurrSource->INF_GetRoundTripTime());
                 FormCtl->EditDelay->Text = IntToStr((__int64)TimingDelay.tv_sec);

                 if (pack->GetPayloadType() == 'B')
                 // paquet indiquant un changement de la taille du buffer
                 {

                       FormCtl->EditStatutReceive->Text = "Changement de Buffer Executé";
                       Paquet = new TPaquet;
                       Paquet->TimeStamp =   pack->GetTimeStamp();
                       Paquet->TimeDiff = abs (Paquet->TimeStamp  - TimeMsecNow());
                       //il faudra rajouter le délai
                       Paquet->PaquetType = 4;
                       Paquet->Buffer =  (*(DataPtr)&0x00FF) |  ((*(DataPtr+1)&0x00ff)<<8);
                       FormCtl->ListInputPrepareRec->AddPaquet(Paquet);

                 };
                 if (pack->GetPayloadType() == 'C')
                 {
                       FormCtl->MemoChat->Lines->Add("you : " + (AnsiString)DataPtr);
                       FormCtl->EditStatutReceive->Text = "on a bien recu un C";
```

```
    };

    if (pack->GetPayloadType() == 'M')
    {

          // on extrait les notes actives
       for (int i = 0 ; i <16 ; i++)
          {
             temp = (__int8)*(DataPtr + i +4);
             for (int j = 0 ; j < 8 ; j++)
             NoteOnRecStatus[(i*8)+j]    = ((temp>>j)&1);
             //on sauve dans le vecteur binaire
          };

         // on sauve le fait qu'elle soit récente
       for (int i = 0 ; i <16 ; i++)
          {
             temp = (__int8)*(DataPtr + i +20);
             for (int j = 0 ; j < 8 ; j++)
               NoteOnRecRecent[(i*8)+j]    = ((temp>>j)&1);
               //on sauve dans le vecteur binaire
          };
          //TEST
       for (int i =0; i< 128 ; i++)
                FormCtl->StatusVectToSend->Notes[i] = NoteOnRecRecent[i];


       Paquet = new TPaquet;
       Paquet->PaquetType = 1;
       Paquet->NumNote = (*(DataPtr + 1));
       Paquet->Message = (msg);
       Paquet->TimeStamp =   pack->GetTimeStamp();
       Paquet->TimeDiff = abs (Paquet->TimeStamp  - TimeMsecNow());

       Paquet->NumSeq = pack->GetExtendedSequenceNumber();
       for (int i =0 ; i < 128; i ++) // on recopie la vue dans paquet
         {
         Paquet->VueStatus[i] = NoteOnRecStatus[i]; //
         Paquet->VueRecent[i] = NoteOnRecRecent[i];

         }

       Paquet->Velocite = (*(DataPtr + 2));

       //ici on va insérer le paquet, il faudrait juste insérer de manière trier
       Continuer = true; Pos = 0;
       while (Continuer & (FormCtl->ListInputPrepareRec->Count() > Pos))
            {
            PaquetBidon =(PtrPaquet)( FormCtl->ListInputPrepareRec->Pos(Pos));
            if (PaquetBidon->NumSeq > Paquet->NumSeq) Continuer = false;
            else Pos++;
            }
       FormCtl->ListInputPrepareRec->InsertPaquet(Paquet, Pos);
   }

    // on a plus besoin du pack
    // donc on l'efface
```

```
            delete pack;

            }
        } while (sess.GotoNextSourceWithData());
        }

        gets(dummybuffer);
FormCtl->EditNbrRec->Text =     IntToStr(NbrPaquetRec);
if (status < 0) FormCtl->EditStatutReceive->Text = "erreur de poll";



////// partie envoi de données ///


if (!(FormCtl->ListOutputPrepareSend->Count()))
{
  Paquet = FormCtl->ListOutputPrepareSend->Pos(0);

  {//il faut calculer l'incrément en ms
  TimeInMsec = TimeMsecNow();
  if (LastSendTimeInMsec == 0) LastSendTimeInMsec = TimeInMsec;  // pour le 1er passage
  status = sess.IncrementTimeStamp((TimeInMsec - LastSendTimeInMsec ));
  }


  //----------------------------------------------------------------------------
  // MIDI DATA TRANSMITTED
  switch(Paquet->PaquetType){
  case 1: //PAQUET MIDI
   {
        msg = (Paquet->Velocite<<16) | (Paquet->NumNote << 8) | (Paquet->Message);
        FormCtl->EditSendMsg->Text = IntToStr(msg);
       if ((Paquet->Message == 144) | (Paquet->Message == 128)) //s'il s'agit d'une note on
ou off
           { // on met à jour la structure locale
           StatusSend[Paquet->NumNote].Status = (Paquet->Message == 144);
           StatusSend[Paquet->NumNote].TimeStamp = TimeInMsec;
           }
        memcpy(&NoteOn , &msg, 4);
      // maintenant il faut remplir les 256 bits pour l'envoi
      //Note on ou off, et récemment ou pas
        for( int i = 0; i < 15 ; i++)
          {
              NoteOn[4+i] = 0 ;      //nettoyer avant usage
              NoteOn[4+16+i] = 0 ;
              for (int j = 0; j < 8 ; j++)
                  {
                  NoteOn[4 + i] = (NoteOn[4+i] | ( (int)StatusSend[(i*8)+j].Status<<j));
                  BoolTemp =  ( TimeInMsec -  StatusSend[(i*8)+j].TimeStamp) < 1000 ;
                                  //delay acceptable

                  NoteOn[4 + 16 + i] = ( NoteOn[4 + 16 + i] |  (int)(BoolTemp<<j) );
                  };
          }

      // il faut incrémenter avant le send packet et pas dans le send packet !!!!
```

```
            status = sess.SendPacket(&NoteOn, 36 ,'M',false, 0);
      }; break ;

    //-----------------------------------------------------------------------------
    // CHAT DATA TRANSMITTED
    case 3: //PAQUET CHAT
      {
      status = sess.SendPacket(Paquet->ChatMessage,sizeof(Paquet-
>ChatMessage),'C',false,0);

      };  break;

    //-----------------------------------------------------------------------------
    // STATE DATA TRANSMITTED
    case 2: //PAQUET STATE
      {

      }; break;
    //-----------------------------------------------------------------------------
    // BUFFER DATA TRANSMITTED
    case 4: //PAQUET BUFFER
      {
      __int16 bidon;
      bidon = Paquet->Buffer;
      msg = (Paquet->Buffer);
      status = sess.SendPacket(&bidon ,2 ,'B',false,0);
      };  break;

   //--------------------------------------------------------------------------

   }; //affichage de qq résultats
   if (status < 0)
      {
            FormCtl->EditStatutSend->Text = "erreur a l'envoi";
      }
   else
      {
            FormCtl->EditStatutSend->Text = "envoi semble correct";
            FormCtl->ListOutputPrepareSend->RemoveFirst();
            NbrPaquetSend++;
            FormCtl->EditNbrSend->Text = NbrPaquetSend;
            LastSendTimeInMsec = TimeInMsec;
      }
     }
   }
}//fin de la fonction

//---------------------------------------------------------------------------------
```

FormCtrl.h

```
#ifndef FormCtrlH
#define FormCtrlH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
```

```
#include <Forms.hpp>
#include <mmsystem.h>
#include <ExtCtrls.hpp>

#include "ListMidi.h"
#include "PERFGRAP.h"
#include <ComCtrls.hpp>
#include "CGAUGES.h"
#include "MidiRead.h"

//---------------------------------------------------------------------------
typedef struct
{
  long int Notes[128]; //

} StatusVect;

class TFormCtl : public TForm
{
__published:    // IDE-managed Components
        TGroupBox *GroupBox2;
        TButton *ButtonReadMidi;
        TEdit *EditByte1;
        TEdit *EditByte2;
        TEdit *EditByte3;
        TEdit *EditCanal;
        TGroupBox *GroupBoxRtp;
        TEdit *EditStatutReceive;
        TEdit *EditStatutSend;
        TEdit *EditStatutConnect;
        TEdit *EditPortLocal;
        TEdit *EditPortDistant;
        TButton *ButtonStartRTP;
        TEdit *EditNbrRec;
        TEdit *EditNbrSend;
        TImage *ImageKeyboardToSend;
        TEdit *EditChat;
        TButton *ButtonChatSend;
        TCheckBox *CheckBoxMidiPlay;
        TMemo *MemoChat;
        TImage *ImageKeyboardToPlay;
        TButton *Button1;
        TButton *ButtonInvertPort;
        TTimer *Timer1;
        TEdit *EditDestIp;
        TEdit *EditSendMsg;
        TCheckBox *CheckBoxMidiCanalOut;
        TEdit *EditMidiCanalOut;
        TCheckBox *CheckBoxTransposeOut;
        TEdit *EditTransposeOut;
        TEdit *EditJitter;
        TEdit *EditDelay;
        TMemo *MemoIpAddr;
        TLabel *LabelListenTo;
        TLabel *LabelSendTo;
        TEdit *EditBufferLocal;
        TCheckBox *CheckBoxCorrection;
```

```
        TEdit *EditSpeed;
        TCGauge *GaugeSpeedLocal;
        TCGauge *GaugeBufferRemote;
        TCheckBox *CheckBoxAutomateLocalBuffer;
        TCGauge *GaugeBufferLocal;
        TLabel *LabelBufferRec;
        TLabel *LabelSpeed;
        TLabel *LabelLocalBuffer;
        TLabel *LabelAttack;
        TLabel *LabelRelease;
        TEdit *EditAttack;
        TEdit *EditRelease;
        TEdit *EditProgram;
        TCheckBox *CheckBoxLocalMIDI;
        TEdit *EditLocalCanal;
        TButton *ButtonSetLocalBuffer;
        TEdit *EditDiffTime;


        void __fastcall ButtonReadMidiClick(TObject *Sender);
        void __fastcall ButtonStartRTPClick(TObject *Sender);
        void __fastcall ButtonChatSendClick(TObject *Sender);
        void __fastcall CheckBoxMidiPlayClick(TObject *Sender);
        void __fastcall EditChatKeyPress(TObject *Sender, char &Key);
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall ButtonInvertPortClick(TObject *Sender);
        void __fastcall Timer1Timer(TObject *Sender);
        void __fastcall ButtonSetLocalBufferClick(TObject *Sender);
        void __fastcall CheckBoxAutomateLocalBufferClick(TObject *Sender);

private: // User declarations
public:
       StatusVect *StatusVectRec;
       StatusVect *StatusVectToSend;

        ThreadMidiList *ListInputPrepareSend; // ce qu'on passe à la gestion d'envoi
        ThreadMidiList *ListOutputPrepareSend;// ce que la gestion d'envoi à traité
        ThreadMidiList *ListInputPrepareRec; // ce qu'on passe à la gestion en reception
        ThreadMidiList *ListLocalMIDI; // ce que l'on doit jouer en local
// MIDI PLAYER SECTION----------------------------------------------------------
        HMIDIOUT    handle;
            int status,i;
            char dummybuffer[1024];

        int NbrPaquet;

        DWORD msg;    // pour le jeu midi
        int note;   // pour le jeu midi
        float Speed; // vitesse en nbr d'évent non simultanés par sec

        int BufferLocal; // soit calculé automatiquement, soit changé dans Edit...
        int BufferRemote;
        TThread *MidiRead;
        TThread *MidiRTPmod;
//        ThreadMidiRead *MidiRead;
        bool MidiReading; // indique si un thread de lecture midi est en cours
        bool SendingRTP;
```

```
//--------------------------------------------------------
        __fastcall TFormCtl(TComponent* Owner);
        void __fastcall DrawStatusVectorCanvas(TCanvas *Canvas,StatusVect *StatusVector);
};
//---------------------------------------------------------------------------
extern PACKAGE TFormCtl *FormCtl;
//---------------------------------------------------------------------------
#endif
```

FormCtrl.cpp

```
#include <stddef.h>
#include <vcl.h>
#include <winsock2.h>
#pragma hdrstop

#include "FormCtrl.h"
#include "PrepareSend.h"
#include "PrepareRec.h"
#include "MidiRtp.h"
#include "MidiGen.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "PERFGRAP"
#pragma link "CGAUGES"
#pragma resource "*.dfm"
TFormCtl *FormCtl;
//---------------------------------------------------------------------------
__fastcall TFormCtl::TFormCtl(TComponent* Owner)
        :TForm(Owner)
{
// creation des threadlist utiles
ListInputPrepareSend = new ThreadMidiList();
ListOutputPrepareSend = new ThreadMidiList();
ListInputPrepareRec = new ThreadMidiList();
ListLocalMIDI = new ThreadMidiList();

ThreadPrepareSend *PrepareSend = new ThreadPrepareSend(false);
ThreadPrepareRec *PrepareRec = new ThreadPrepareRec(false);


StatusVectRec = new StatusVect;//etat reçu
StatusVectToSend = new StatusVect;//etat émis
//on nettoye les vecteurs
for (int i=0 ; i < 128 ; i++)
        {
        StatusVectRec->Notes[i] = false;
        StatusVectToSend->Notes[i] = false;
        }

// cohérence affichage et variable
BufferLocal = StrToInt(EditBufferLocal->Text);
```

```
MidiReading = false; //lecture Midi
SendingRTP = false; //envoi rtp
}
//---------------------------------------------------------------------------
extern PACKAGE TFormCtl *TFormCtl;

//---------------------------------------------------------------------------
void __fastcall TFormCtl::ButtonReadMidiClick(TObject *Sender)
{
MidiReading = !MidiReading;
if (MidiReading)
  MidiRead = new ThreadMidiRead(false);
}
//---------------------------------------------------------------------------

void __fastcall TFormCtl::ButtonStartRTPClick(TObject *Sender)
{
if (!SendingRTP) //le button lance ou stoppe le module RTP
  MidiRTPmod = new MidiRTP(false);
 else MidiRTPmod->Terminate();

SendingRTP = ! SendingRTP;

}
//---------------------------------------------------------------------------
//-------------------------------------------------------------------------
//fonction réalisant l'affichage du clavier sur le canvas passé en param
void __fastcall TFormCtl::DrawStatusVectorCanvas(TCanvas *Canvas,
                                StatusVect *StatusVector)

{

int CanvasHeight;
int CanvasWidth;
int BlackKeyWidth;
int WhiteKeyWidth;
int NbrOctave = 5;
int shift;

bool blanche ; // indique si on a à faire à une note blanche ou noire
bool NoteOn; // indique si la touche est enfoncée ou non


TRect Rect;
TRect RectTouche;

CanvasHeight = Canvas->ClipRect.Height();
CanvasWidth = Canvas->ClipRect.Width();


Rect = Canvas->ClipRect;

WhiteKeyWidth = (CanvasWidth / (7*NbrOctave));
BlackKeyWidth = (WhiteKeyWidth / 2);


//--------------------- on va dessiner les blanches ----------------------------------
```

```
for (int i = 0 ; i < NbrOctave ; i++)     {
  for (int j = 0 ; j < 12 ; j++) // 12 touches par octaves
      {
      NoteOn = StatusVector->Notes[36 + j+ (i * 12)];
      switch(j){
        case 0: blanche = true; shift =0; break;
        case 2: blanche = true; shift =1;break;
        case 4:blanche = true; shift =2;break;
        case 5:blanche = true; shift =3;break;
        case 7:blanche = true; shift =4;break;
        case 9:blanche = true; shift =5;break;
        case 11:blanche = true; shift =6;break;
        default : blanche = false;
         }

        if (blanche)
          {
          RectTouche.left = Canvas->ClipRect.Left + ((WhiteKeyWidth * 7) * i)
                                    + (WhiteKeyWidth * shift);
          RectTouche.right = RectTouche.left + WhiteKeyWidth;
          RectTouche.top = Canvas->ClipRect.Top;
          RectTouche.bottom = Canvas->ClipRect.bottom;
          Canvas->Brush->Color = clBlack;
          Canvas->Rectangle(RectTouche.left,RectTouche.top,
                    RectTouche.right,RectTouche.bottom);
          if (NoteOn) //si la note est active : dessin rouge
              {
              Canvas->Brush->Color = clRed; //
              Canvas->Rectangle(RectTouche.left,
                     RectTouche.top,RectTouche.right,RectTouche.bottom);
              }
          else
              {      //sinon en blanc
              Canvas->Brush->Color = clWhite;
              Canvas->Rectangle(RectTouche.left,RectTouche.top,
                       RectTouche.right,RectTouche.bottom);
              }
          }
       }
}
//---------------------- on va maintenant supperposer les touches noires
for (int i = 0 ; i < NbrOctave ; i++)     {
  for (int j = 0 ; j < 12 ; j++) // 12 touches par octaves
      {
      NoteOn = StatusVector->Notes[36 + j + (i * 12)];
      switch(j){
        case 1: blanche = false; shift =0; break;
        case 3: blanche = false; shift =1;break;
        case 6:blanche = false; shift =3;break;
        case 8:blanche = false; shift =4;break;
        case 10:blanche = false; shift =5;break;
        default : blanche = true;
         }

        if (!blanche)
          {
          RectTouche.left = Canvas->ClipRect.left + (WhiteKeyWidth/2)+
```

```
          (WhiteKeyWidth/3)+((WhiteKeyWidth * 7) * i) + (WhiteKeyWidth * shift);

          RectTouche.right = RectTouche.left + WhiteKeyWidth -(WhiteKeyWidth/3);
          RectTouche.top = Canvas->ClipRect.Top;
          RectTouche.bottom = Canvas->ClipRect.Top + 5*(CanvasHeight /7) ;
          Canvas->Brush->Color = clBlack;
//(int X1, int Y1, int X2, int Y2); x1 , y1 = upper left
          Canvas->Rectangle(RectTouche.left,RectTouche.top,
          RectTouche.right,RectTouche.bottom);

          if (NoteOn)
             {
             Canvas->Brush->Color = clRed;
             Canvas->Rectangle(RectTouche.left,RectTouche.top,
               RectTouche.right,RectTouche.bottom);
             }
          else
             {
             Canvas->Brush->Color = clBlack;
             Canvas->Rectangle(RectTouche.left,RectTouche.top,RectTouche.right,
                                    RectTouche.bottom);
             }
          }
       }

}
}
//---------------------------------------------------------------------------
void __fastcall TFormCtl::ButtonChatSendClick(TObject *Sender)
{
PtrPaquet Paquet;
Paquet = new TPaquet;
Paquet->PaquetType = 3; // 3 = chat msg

EditChat->GetTextBuf(Paquet->ChatMessage, 29);//temporairement 29 char
MemoChat->Lines->Add("I : "+EditChat->Text);
EditChat->Text = "";
ListInputPrepareSend->AddPaquet(Paquet);
}
//---------------------------------------------------------------------------
void __fastcall TFormCtl::CheckBoxMidiPlayClick(TObject *Sender)
//activation ou désactivation du playback midi midi
{
if (CheckBoxMidiPlay->Checked)
  {
  midiOutOpen(&handle , (UINT)-1, 0, 0, CALLBACK_NULL);
  }
else
  {
  midiOutClose(handle);
  }
}

//---------------------------------------------------------------------------
void __fastcall TFormCtl::EditChatKeyPress(TObject *Sender, char &Key)
{
if (Key ==13)
```

```
  ButtonChatSendClick(Sender);//touche enter pour l'envoi
}
//---------------------------------------------------------------------------

void __fastcall TFormCtl::Button1Click(TObject *Sender)
{
MidiGenerator *MidiGen = new MidiGenerator(false);//création d'un générateur midi
}
//---------------------------------------------------------------------------

void __fastcall TFormCtl::ButtonInvertPortClick(TObject *Sender)
{//permute les 2 ports indiqués
AnsiString temp;
temp = EditPortLocal->Text;
EditPortLocal->Text=EditPortDistant->Text;
EditPortDistant->Text=temp;
}
//---------------------------------------------------------------------------

void __fastcall TFormCtl::Timer1Timer(TObject *Sender)
{
//mise à jour régulière de l'affichage
DrawStatusVectorCanvas(ImageKeyboardToPlay->Canvas, StatusVectRec);
DrawStatusVectorCanvas(ImageKeyboardToSend->Canvas, StatusVectToSend);
//speed est en seconde mais on va l'afficher en minutes
EditSpeed->Text = FloatToStrF((Speed*60.0),0,4,5);
if (CheckBoxAutomateLocalBuffer->Checked)
  EditBufferLocal->Text = IntToStr(BufferLocal);
//vitesse max à 1000 event par minutes
GaugeSpeedLocal->Progress =   min(1000,((int)(Speed*60)));
GaugeBufferLocal->Progress = min(1000,(BufferLocal )); // max = 150ms
GaugeBufferRemote->Progress = min(1000,(BufferRemote)); // max = 150ms

// on va envoyer un paquet pour signaler que l'info buffer à changée
}
//---------------------------------------------------------------------------
void __fastcall TFormCtl::ButtonSetLocalBufferClick(TObject *Sender)
{//taille du buffer local indiqué manuellement
  BufferLocal = StrToInt(EditBufferLocal->Text);
}
//---------------------------------------------------------------------------

void __fastcall TFormCtl::CheckBoxAutomateLocalBufferClick(TObject *Sender)
{//passage en mode buffer automatique
ButtonSetLocalBuffer->Enabled = !CheckBoxAutomateLocalBuffer->Checked;
}
//---------------------------------------------------------------------------
```

MidiRTP.h

```
#include <stdio.h>
#include <stdlib.h>
#ifndef WIN32
#define WIN32
#endif
```

```
#include "rtpsession.h"
#include "rtppacket.h"
//---------------------------------------------------------------------------
struct StatusNotDef
{
  bool Status; // true = noteOn , false = NoteOff
  unsigned long TimeStamp;
  long int NumSeq;
  int Velocite;
};


class MidiRTP : public TThread
{
private:
protected:

public:
        WSADATA wsaData;
            RTPSession sess;
            int portbase;
            unsigned long destip;
            int destport;
            char ipstr[256];
            int status,i;
            char dummybuffer[1024];

        int NbrPaquetRec; // nombre de paquet recu
        int NbrPaquetSend; // nombre de paquet émis



        __fastcall MidiRTP(bool CreateSuspended);
        void __fastcall Execute();
        void __fastcall SendReceive();
        StatusNotDef StatusSend[128]; // ça c'est ce qu'on a envoyé, le jeu local


};
//---------------------------------------------------------------------------
```

MidiRTP.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include <dos.h>

#include "rtpsession.h"
#include "rtppacket.h"
#include "ListMidi.h"
#include "MidiRtp.h"
#include "FormCtrl.h"
```

```
#include "FonctionUtile.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------

//   Important: Methods and properties of objects in VCL can only be
//   used in a method called using Synchronize, for example:
//
//      Synchronize(UpdateCaption);
//
//   where UpdateCaption could look like:
//
//      void __fastcall Unit1::UpdateCaption()
//      {
//        Form1->Caption = "Updated in a thread";
//      }
//---------------------------------------------------------------------------

__fastcall MidiRTP::MidiRTP(bool CreateSuspended)
        : TThread(CreateSuspended)
{
for (int i = 0 ; i < 128 ; i++)  // on met à jour la liste des temps
        {
              StatusSend[i].TimeStamp = 0;
              StatusSend[i].Status  = false;
        }

}
//---------------------------------------------------------------------------
void __fastcall MidiRTP::Execute()
{
bool errorRTP = false;

if(WSAStartup(0x101, &wsaData))
    {
        FormCtl->EditStatutConnect->Text =  "Unable to initialize WinSock library.\n";
        exit;
    }

        sess.Destroy();

        portbase = StrToInt(FormCtl->EditPortLocal->Text);

        destip = inet_addr(FormCtl->EditDestIp->Text.c_str());
            if (destip == INADDR_NONE)
            {
                        FormCtl->EditStatutConnect->Text = "Bad IP address specified";
              exit;

            }

            // The inet_addr function returns a value in network byte order, but
            // we need the IP address in host byte order, so we use a call to
            // ntohl
            destip = ntohl(destip);


            destport = StrToInt(FormCtl->EditPortDistant->Text); //PORT DE DESTINATION
```

```
    hostent *p;
    int i;
    char s[128];
    char *p2;
    //Get the computer name
    gethostname(s, 128);
    p = gethostbyname(s);
    FormCtl->MemoIpAddr->Lines->Add(p->h_name);


//    Get the IpAddresses
i = 0;
while (p->h_addr_list[i] != NULL)
    {
    p2 = inet_ntoa(*((in_addr *)p->h_addr_list[i]));
    FormCtl->MemoIpAddr->Lines->Add(p2);
    i++;
    }
            /*
              Now, we'll create a RTP session, set the destination, send some
              packets and poll for incoming data.
            */

            status = sess.Create(portbase);
        if (status < 0) errorRTP = true;
        sess.SetTimestampUnit(1.0/1000.0);



            status = sess.AddDestination(destip,destport);
        if (status < 0) errorRTP = true;
        if (!errorRTP)
          {
          SendReceive(); // on lit on ecrit alternativement.
          FormCtl->EditStatutConnect->Text = "RTP Initialisé";
          }
        else
         FormCtl->EditStatutConnect->Text = "Erreur RTP";
}
//---------------------------------------------------------------------------
void __fastcall MidiRTP::SendReceive()

{

char buffer[32];
char *DataPtr;

struct time Now;
unsigned long  TimeInMsec = 0 ;
unsigned long  LastSendTimeInMsec = 0 ;

__int8 temp;
__int8 NoteOn[36];
int Pos;
int Delay = 0;
```

```
bool NoteOnRecStatus[256]; // 4 premier bytes sont le message ,
//les 32 bytes suivants représentes les statuts : actifs et récent
bool NoteOnRecRecent[256];
bool BoolTemp;
bool Continuer;

DWORD msg;
AnsiString ChatMsg;

RTPSourceData *CurrSource;

PtrPaquet Paquet, PaquetBidon;
timeval TimingDelay;

NbrPaquetRec = 0;
NbrPaquetSend =0;


   while (FormCtl->SendingRTP)//
   {
   Sleep(2);

   status = sess.PollData();

         // check incoming packets

   if (sess.GotoFirstSourceWithData())
   {

    CurrSource = sess.GetCurrentSourceInfo();
    CurrSource->SetTimestampUnit(1.0/1000.0); // les incrementations de timestamp se font
par 1ms

         do
         {
               RTPPacket *pack;

               while ((pack = sess.GetNextPacket()) != NULL)
               {
                 // You can examine the data here
                 NbrPaquetRec++ ;
                 msg =  (*pack->GetPayload());

                 DataPtr = pack->GetPayload();


                 FormCtl->EditJitter->Text =   IntToStr(CurrSource->INF_GetJitter());
                 TimingDelay = (CurrSource->INF_GetRoundTripTime());
                 FormCtl->EditDelay->Text = IntToStr((__int64)TimingDelay.tv_sec);

                 if (pack->GetPayloadType() == 'B')
                 // paquet indiquant un changement de la taille du buffer
                 {

                       FormCtl->EditStatutReceive->Text = "Changement de Buffer Executé";
                       Paquet = new TPaquet;
                       Paquet->TimeStamp =   pack->GetTimeStamp();
```

```
        Paquet->TimeDiff = abs (Paquet->TimeStamp  - TimeMsecNow());
        //il faudra rajouter le délai
        Paquet->PaquetType = 4;
        Paquet->Buffer =  (*(DataPtr)&0x00FF) |  ((*(DataPtr+1)&0x00ff)<<8);
        FormCtl->ListInputPrepareRec->AddPaquet(Paquet);

};
if (pack->GetPayloadType() == 'C')
{
        FormCtl->MemoChat->Lines->Add("you : " + (AnsiString)DataPtr);
        FormCtl->EditStatutReceive->Text = "on a bien recu un C";
};
if (pack->GetPayloadType() == 'M')
{

          // on extrait les notes actives
       for (int i = 0 ; i <16 ; i++)
          {
             temp = (__int8)*(DataPtr + i +4);
             for (int j = 0 ; j < 8 ; j++)
             NoteOnRecStatus[(i*8)+j]    = ((temp>>j)&1);
             //on sauve dans le vecteur binaire
          };

        // on sauve le fait qu'elle soit récente
       for (int i = 0 ; i <16 ; i++)
          {
             temp = (__int8)*(DataPtr + i +20);
             for (int j = 0 ; j < 8 ; j++)
               NoteOnRecRecent[(i*8)+j]    = ((temp>>j)&1);
               //on sauve dans le vecteur binaire
          };
          //TEST
       for (int i =0; i< 128 ; i++)
               FormCtl->StatusVectToSend->Notes[i] = NoteOnRecRecent[i];


       Paquet = new TPaquet;
       Paquet->PaquetType = 1;
       Paquet->NumNote = (*(DataPtr + 1));
       Paquet->Message = (msg);
       Paquet->TimeStamp =   pack->GetTimeStamp();
       Paquet->TimeDiff = abs (Paquet->TimeStamp  - TimeMsecNow());

       Paquet->NumSeq = pack->GetExtendedSequenceNumber();
       for (int i =0 ; i < 128; i ++) // on recopie la vue dans paquet
         {
         Paquet->VueStatus[i] = NoteOnRecStatus[i]; //
         Paquet->VueRecent[i] = NoteOnRecRecent[i];

         }



       Paquet->Velocite = (*(DataPtr + 2));

       //ici on va insérer le paquet, il faudrait juste insérer de manière trier
```

```
                    Continuer = true; Pos = 0;
                    while (Continuer & (FormCtl->ListInputPrepareRec->Count() > Pos))
                         {
                         PaquetBidon =(PtrPaquet)( FormCtl->ListInputPrepareRec->Pos(Pos));
                         if (PaquetBidon->NumSeq > Paquet->NumSeq) Continuer = false;
                         else Pos++;
                         }
                    FormCtl->ListInputPrepareRec->InsertPaquet(Paquet, Pos);
               }

               // on a plus besoin du pack
               // donc on l'efface
               delete pack;

              }
         } while (sess.GotoNextSourceWithData());
         }

         gets(dummybuffer);
FormCtl->EditNbrRec->Text =     IntToStr(NbrPaquetRec);
if (status < 0) FormCtl->EditStatutReceive->Text = "erreur de poll";

////// partie envoi de données ///////////////////////////////////////////////////////

if (!(FormCtl->ListOutputPrepareSend->Count() == 0))
{
   Paquet = FormCtl->ListOutputPrepareSend->Pos(0);

   {//il faut calculer l'incrément en ms
   TimeInMsec = TimeMsecNow();
   if (LastSendTimeInMsec == 0) LastSendTimeInMsec = TimeInMsec;  // pour le 1er passage
   status = sess.IncrementTimeStamp((TimeInMsec - LastSendTimeInMsec ));
   }


   //---------------------------------------------------------------------------
   // MIDI DATA TRANSMITTED
   switch(Paquet->PaquetType){
   case 1: //PAQUET MIDI
    {
         msg = (Paquet->Velocite<<16) | (Paquet->NumNote << 8) | (Paquet->Message);
         FormCtl->EditSendMsg->Text = IntToStr(msg);
         if ((Paquet->Message == 144) | (Paquet->Message == 128)) //s'il s'agit d'une note on
ou off
              { // on met à jour la structure locale
              StatusSend[Paquet->NumNote].Status = (Paquet->Message == 144);
              StatusSend[Paquet->NumNote].TimeStamp = TimeInMsec;
              }

         memcpy(&NoteOn , &msg, 4);

       // maintenant il faut remplir les 256 bits pour l'envoi
       //Note on ou off, et récemment ou pas
         for( int i = 0; i < 15 ; i++)
           {
              NoteOn[4+i] = 0 ;      //nettoyer avant usage
              NoteOn[4+16+i] = 0 ;
```

```
            for (int j = 0; j < 8 ; j++)
                {
                NoteOn[4 + i] = (NoteOn[4+i] | ( (int)StatusSend[(i*8)+j].Status<<j));
                BoolTemp =  ( TimeInMsec -  StatusSend[(i*8)+j].TimeStamp) < 1000 ;
                                //delay acceptable

                NoteOn[4 + 16 + i] = ( NoteOn[4 + 16 + i] |  (int)(BoolTemp<<j) );
                };
        }
      // il faut incrémenter avant le send packet et pas dans le send packet !!!!

       status = sess.SendPacket(&NoteOn, 36 ,'M',false, 0);

    }; break ;

   //---------------------------------------------------------------------------
   // CHAT DATA TRANSMITTED
   case 3: //PAQUET CHAT
    {
    status = sess.SendPacket(Paquet->ChatMessage,sizeof(Paquet-
>ChatMessage),'C',false,0);
    };  break;

   //---------------------------------------------------------------------------
   // STATE DATA TRANSMITTED
   case 2: //PAQUET STATE
    {
    }; break;

   //---------------------------------------------------------------------------
   // BUFFER DATA TRANSMITTED
   case 4: //PAQUET BUFFER
    {
    __int16 bidon;
    bidon = Paquet->Buffer;

    msg = (Paquet->Buffer);
    status = sess.SendPacket(&bidon ,2 ,'B',false,0);

    };  break;

  //-------------------------------------------------------------------------

  }; //affichage de qq résultats
  if (status < 0)
    {
       FormCtl->EditStatutSend->Text = "erreur a l'envoi";
    }
  else
    {
       FormCtl->EditStatutSend->Text = "envoi semble correct";
       FormCtl->ListOutputPrepareSend->RemoveFirst();
       NbrPaquetSend++;
       FormCtl->EditNbrSend->Text = NbrPaquetSend;
       LastSendTimeInMsec = TimeInMsec;
    }
   }
```

```
   }
}//fin de la fonction
//---------------------------------------------------------------------------
```

PrepareRec.h

```
//---------------------------------------------------------------------------

#ifndef PrepareRecH
#define PrepareRecH
//---------------------------------------------------------------------------
#include <Classes.hpp>
//---------------------------------------------------------------------------
struct StatusNoteDef
{
  bool Status; // true = noteOn , false = NoteOff
  unsigned long TimeStamp;
  long int NumSeq;
  int Velocite;
};
//---------------------------------------------------------------------------

class ThreadPrepareRec : public TThread
{
private:
protected:
        void __fastcall Execute();
public:
        __fastcall ThreadPrepareRec(bool CreateSuspended);
        void __fastcall Prepare();
        void __fastcall PlayNote(PtrPaquet Paquet, bool localMidi);



// partie de donnée pour la correction des erreurs.
        bool FirstNote;
        int Delay;      // pour les tests on lui donne la valeur du forwarder
        unsigned long TimeDiff;
        // la différence entre l'horloge du timestamp et celle du pc. //
        unsigned long LastNumSeq;  // dernier numéro de sequence auquel on a eu à faire

        StatusNoteDef StatusNotes[128];  //ça c'est ce qui a été recu
        int BufferLocal;
        int BufferRemote;
        bool CorrectionStatus;
        bool NoBufferLocal;
        bool NoBuffer;
        TList *CorrectionQueue;
};
//---------------------------------------------------------------------------
#endif
```

PrepareRec.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop
```

```
#pragma package(smart_init)
#include <mmsystem.h>
#include <assert.h>


#include "FormCtrl.h"
#include "ListMidi.h"
#include "PrepareRec.h"
#include "FonctionUtile.h"
//---------------------------------------------------------------------------

//   Important: Methods and properties of objects in VCL can only be
//   used in a method called using Synchronize, for example:
//
//      Synchronize(UpdateCaption);
//
//   where UpdateCaption could look like:
//
//      void __fastcall Unit1::UpdateCaption()
//      {
//        Form1->Caption = "Updated in a thread";
//      }
//---------------------------------------------------------------------------

__fastcall ThreadPrepareRec::ThreadPrepareRec(bool CreateSuspended)
        : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall ThreadPrepareRec::Execute()
{

FirstNote = True; // la note que l'on recevra sera la première
Delay = 0; // 30 MS
NoBufferLocal = false;
NoBuffer = false;
BufferLocal = FormCtl->BufferLocal;
CorrectionStatus = True;      // soit on corrige soit on fait de l'envoi simple
CorrectionQueue = new TList;   // on prépare la liste pour le correcteur ;

for (int i=0; i<128; i++)              // on fait le nettoyage de la structure
{
        StatusNotes[i].Status = False;
        StatusNotes[i].TimeStamp = 0;
        StatusNotes[i].NumSeq = 0;
}
Prepare();
}
//------------------------------------------------------------------------



//
void __fastcall ThreadPrepareRec::Prepare()
{
```

```
PtrPaquet Paquet;
PtrPaquet PaquetBidon;
unsigned long msg;
int NumNote;
unsigned long TimeNow;
unsigned long OldTimeNow;
unsigned long OldTimeStamp;
bool Continuer = true;
bool LocalMidi = False;
int Pos = 0;
int IndList = 0;
bool FirstNote = true;
int cpt;

cpt=0;
while(true)
   {

     Sleep(3);
     TimeNow = TimeMsecNow();
     BufferLocal = FormCtl->BufferLocal;
     BufferRemote = FormCtl->BufferRemote;
     CorrectionStatus = FormCtl->CheckBoxCorrection->Checked;
     NoBuffer = !FormCtl->CheckBoxCorrection->Checked;

     TimeNow = TimeMsecNow();

     if (FormCtl->ListInputPrepareRec->Count() != 0)
       {
          Paquet = FormCtl->ListInputPrepareRec->Pos(0);


         //on calcule la différence de temps entre l'emetteur et le récepteur
          if (FirstNote)
              {
                FirstNote = False;
                TimeDiff = abs (Paquet->TimeStamp  - TimeNow);
                // on calcule la différence entre horloge
                FormCtl->EditDiffTime->Text = IntToStr(TimeDiff);
                //il faudra envoyer dès la connexion un certain nombre
                //de paquet qui permette à RTP de calculer le Delay
                //il faudra ptet recalculer pour que ce soit un min
              }
           if (NoBuffer |(abs(( abs(Paquet->TimeStamp - TimeNow))
           - Paquet->TimeDiff)  > (BufferRemote)) )
                {
     // ce n'est qu'a ce moment que l'on peut enlever les paquets de la liste
                         FormCtl->ListInputPrepareRec->RemoveFirst();
          // TRAITEMENT DES PAQUET DE TYPE MIDI MESSAGE NOTE ON NOTE OFF
                         if (Paquet->PaquetType == 1)   //midi
                       {
                        if (/*CorrectionStatus & */((Paquet->Message==128) |(Paquet-
>Message==144)) )
                               {

                                    if ((StatusNotes[Paquet->NumNote].Status != (Paquet->Message
== 144)
```

```
                  & (StatusNotes[Paquet->NumNote].NumSeq < Paquet-
>NumSeq)))
                     {
                     LastNumSeq = Paquet->NumSeq;
                     PlayNote(Paquet, false);
                     }

                 for (int i = 0 ; i < 128 ; i++)
                 // pour chaque note de la vue

                  if (Paquet->TimeStamp > StatusNotes[i].TimeStamp) // c'est un
paquet récent
                   if (StatusNotes[i].Status != Paquet->VueStatus[i])
                   // si c'est un paquet différent de ce qui y est
                   {
                    Paquet->NumNote=i;
                    Paquet->Velocite = 100;
                    if (!Paquet->VueStatus[i]) // C'est un Noteoff
                               {
                                 Paquet->Message  = 128;
                                 PlayNote(Paquet, false);
                               };
                    if (Paquet->VueStatus[i])
                              //C'est un NoteOn
                              //traitement juste si récent
                              if (Paquet->VueRecent[i])
                                 {
                                     PlayNote(Paquet, false);
                                     Paquet->Message  = 144;
                                 }
                        }

               }
                    else
                         {
                                  LastNumSeq = Paquet->NumSeq;
                                  PlayNote(Paquet, false);
                         };
                  }
                 // TRAITEMENT DES PAQUET DE TYPE ETAT
          if (Paquet->PaquetType == 2)   //state
                 {

                 }

                 // TRAITEMENT DES PAQUET DE TYPE CHAT
          if (Paquet->PaquetType == 3)   //chat
                 {

                 }
         if (Paquet->PaquetType == 4)   // buffer change
                 {
                 BufferRemote = Paquet->Buffer;
                 FormCtl->BufferRemote=BufferRemote;

                 }
```

```
                }
        }


        // TRAITEMENT DU JEU LOCAL
         if (FormCtl->ListLocalMIDI->Count() != 0)
        {
           Paquet = FormCtl->ListLocalMIDI->Pos(0);
            if (NoBufferLocal | (( abs(TimeNow - Paquet->TimeStamp ) > (BufferLocal)) ))
                {
                        FormCtl->ListLocalMIDI->RemoveFirst();
                        if (Paquet->PaquetType == 1)   //midi , normalement c'est sur
                                PlayNote(Paquet, true);
                }
        }



}

}
//---------------------------------------------------------------------------
void __fastcall ThreadPrepareRec::PlayNote(PtrPaquet Paquet, bool LocalMidi)
{
unsigned long msg;
int NumNote;

// on vérifie d'abord qu'on ne va pas jouer une note dans le meme statut que celle que l'on
veut jouer
if  (StatusNotes[Paquet->NumNote].Status != (Paquet->Message == 144) | LocalMidi    )
{
   if (!LocalMidi) //partie à executer uniquement pour le jeu distant
        {
        StatusNotes[Paquet->NumNote].Status = (Paquet->Message == 144);
        StatusNotes[Paquet->NumNote].NumSeq = Paquet->NumSeq;
        StatusNotes[Paquet->NumNote].TimeStamp = Paquet->TimeStamp;
         }

        // on vérifier s'il faut transposer et on le fait éventuellement
        NumNote = Paquet->NumNote;
        if (FormCtl->CheckBoxTransposeOut->Checked)
              NumNote = NumNote + StrToInt(FormCtl->EditTransposeOut->Text) ;
        //-----------------------------------------------------------
        msg = (Paquet->Velocite<<16) | (NumNote << 8) | (Paquet->Message);
        if (Paquet->Message == 144 & !LocalMidi)
           FormCtl->StatusVectRec->Notes[Paquet->NumNote] = true;
        else   FormCtl->StatusVectRec->Notes[Paquet->NumNote] = false;

        // on verifie s'il faut changer le canal de sortie et on le fait eventuellement
        if (FormCtl->CheckBoxMidiCanalOut->Checked & !LocalMidi)
              msg = msg + StrToInt(FormCtl->EditMidiCanalOut->Text) - 1;
        //-------------------------------------------------------
        if (FormCtl->CheckBoxLocalMIDI->Checked & LocalMidi)
              msg = msg + StrToInt(FormCtl->EditLocalCanal->Text) - 1;
        midiOutShortMsg(FormCtl->handle, msg);  // on joue la note
```

```
        }
}
//---------------------------------------------------------------------------
```

ListMidi.h

```
//---------------------------------------------------------------------------
#ifndef ThreadListMidiH
#define ThreadListMidiH

typedef struct AList
{
  int PaquetType;  // 1 = midi ; 2 = state; 3 = chat;
  unsigned long TimeStamp;
  unsigned long NumSeq;
  unsigned long TimeDiff;
  int NumNote;
  int Message;
  int Canal ;
  char ChatMessage[30];
  unsigned long Buffer;
  int Velocite;
  bool VueStatus[128];
  bool VueRecent[128];

} TPaquet;

typedef TPaquet* PtrPaquet;


class ThreadMidiList
{
private: // User declarations
public:         // User declarations
        int bidon;
        TThreadList *MyList;
        ThreadMidiList();
        AddPaquet(PtrPaquet Paquet);
        InsertPaquet(PtrPaquet Paquet, int i);

        int Count();
        PtrPaquet Pos(int i);
        RemoveFirst();
        Remove(int x);
        bool status[128];
        bool recent[128];
};



//---------------------------------------------------------------------------
#endif
```

ListMidi.cpp

```
//---------------------------------------------------------------------------
```

```
#include <vcl.h>
#pragma hdrstop

#include "ListMidi.h"

//---------------------------------------------------------------------------
ThreadMidiList::ThreadMidiList()
{
MyList = new TThreadList;

}

//---------------------------------------------------------------------
int ThreadMidiList::Count()
{
int temp;
temp = MyList->LockList()->Count;
MyList->UnlockList();
return temp;
}

//---------------------------------------------------------------------



//---------------------------------------------------------------------
ThreadMidiList::InsertPaquet(PtrPaquet Paquet, int i)
{
MyList->LockList()->Insert(i,Paquet);
MyList->UnlockList();
}
//--------------------------------------------------------------------


ThreadMidiList::AddPaquet(PtrPaquet Paquet)
{

MyList->Add(Paquet);


}
//---------------------------------------------------------------------
PtrPaquet ThreadMidiList::Pos(int i)
{
PtrPaquet PaquetTmp;
PaquetTmp = (PtrPaquet)(MyList->LockList()->Items[i]);
MyList->UnlockList();
return  PaquetTmp;
}
//---------------------------------------------------------------------


ThreadMidiList::RemoveFirst()
{
MyList->LockList()->Delete(0);
MyList->UnlockList();
```

```
}
//---------------------------------------------------------------------------
ThreadMidiList::Remove(int x)
{
MyList->LockList()->Delete(x);
MyList->UnlockList();
}
//---------------------------------------------------------------------------

#pragma package(smart_init)
```

FonctionUtile.h

```
//---------------------------------------------------------------------------
#ifndef FonctionUtileH
#define FonctionUtileH
//---------------------------------------------------------------------------
#endif

typedef struct TimePack

{
  unsigned long  Time;
  bool NoteOn;

} TimePaquet;


typedef TimePaquet* PtrTimePaquet;


unsigned long TimeMsecNow();

int BufferResizer(int Speed ,int Attack, int Release,  int Delay, int Gigue,int Quality);

//---------------------------------------------------------------------------

class SpeedCalculator
{
private:
        unsigned long  TimeOfLastEvent;
        int NbrEvent;
        int SpeedNow;
        int TimeInterval ; // indique le temps nécessaire pour calculer la vitesse;
        TList *EventInInterval;
        TimePaquet *ZeTimePaquet;
        void Clean();


protected:

public:

        SpeedCalculator::SpeedCalculator(); //constructor
        void SpeedCalculator::NotifyOn(bool NoteOn); // indique qu'une nouvelle note on là
        float SpeedCalculator::Speed();  // renvoi la vitesse actuelle
        void SpeedCalculator::ChangeTimeIntervalle(int second);
```

```
};

//---------------------------------------------------------------------------
```

FonctionUtile.cpp

```
//---------------------------------------------------------------------------
#include <vcl.h>
#include <dos.h>

#pragma hdrstop

#include "FonctionUtile.h"
#include <dos.h>
//---------------------------------------------------------------------------
#pragma package(smart_init)
unsigned long TimeMsecNow()
//renvoi le temps actuel en millisecond (avec une précision au 1/100 de sec
{
struct  time Now;
unsigned long TimeInMsec10;
gettime(&Now);   // la ligne suivante nous donne le temps en dizaine de msec.
TimeInMsec10 =(Now.ti_hour * 60 * 60 * 100) + (Now.ti_min * 60 * 100)
+ (Now.ti_sec * 100) + (Now.ti_hund);
return (TimeInMsec10 * 10) ;
}
//---------------------------------------------------------------------------
int BufferResizer(int Speed ,int Attack, int Release,  int Delay, int Gigue,int Quality)
{
// accept all values in Msec
// quality value is from 1 to 10 -> 10 Reduce the BufferSize
double DefaultBuffer;
DefaultBuffer= ((-6.0/14.0)*((Speed*60.0)/2.5))+120.0  ;
DefaultBuffer = DefaultBuffer + Attack + Release;
DefaultBuffer = max(DefaultBuffer, 10.0);
DefaultBuffer = min(DefaultBuffer, 500.0);


return DefaultBuffer;
}

//---------------------------------------------------------------------------
SpeedCalculator::SpeedCalculator() //constructor
{
NbrEvent  = 0;
SpeedNow = 0;
TimeInterval = 5 * 1000; // par défaut la vitesse est calculée pour une période de deux
secondes
EventInInterval = new TList;
}
//---------------------------------------------------------------------------
void SpeedCalculator::NotifyOn(bool NoteOn) // indique qu'une nouvelle note on est arrivé
{
unsigned long Time;
unsigned long TimeNow;
```

```
ZeTimePaquet = new TimePaquet;
TimeNow = TimeMsecNow();
if ((TimeNow - TimeOfLastEvent) > 40) //on considère deux évenements à moins de 40 ms
simultanés
  {
    ZeTimePaquet->Time = TimeMsecNow();
    ZeTimePaquet->NoteOn = NoteOn;
    EventInInterval->Add(ZeTimePaquet);
    ++NbrEvent;
    TimeOfLastEvent = TimeNow; // on ne met pas à jour sinon on ne met pas à jour pour ête
sur que
                               // on va traiter les suivants
  }

}
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void SpeedCalculator::Clean()
{

float value;
unsigned long TempSpeed;
unsigned long TimeNow;
int NbrInList;
int Pos=0;

TimePaquet *TempTimePaquet;
TimeNow = TimeMsecNow();
while (Pos < EventInInterval->Count)
        {
        TempTimePaquet =(PtrTimePaquet)EventInInterval->Items[Pos];
        // on prend le paquet à la position courante
        if ( TimeNow > (TempTimePaquet->Time  + TimeInterval))
          {
          EventInInterval->Delete(Pos);
//          delete(TempTimePaquet);
          }
        else Pos++; // si on efface pas un paquet on passe à l'élément suivant
        }
}
//---------------------------------------------------------------------------
float SpeedCalculator::Speed()  // renvoi la vitesse actuelle
{
Clean();
return (EventInInterval->Count / (TimeInterval/1000.0));
// on renvoye le nombre d'intervalle par seconde
}
//---------------------------------------------------------------------------
void SpeedCalculator::ChangeTimeIntervalle(int second)
{
TimeInterval = second * 1000;
}
//---------------------------------------------------------------------------
```

PrepareSend.h

```
//---------------------------------------------------------------------------
```

```
#ifndef PrepareSendH
#define PrepareSendH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <FonctionUtile.h>
//---------------------------------------------------------------------------
class ThreadPrepareSend : public TThread
{
private:

protected:
        void __fastcall Execute();
public:
        int NbrNotesForSpeed;
        SpeedCalculator *SpeedCalc;
        __fastcall ThreadPrepareSend(bool CreateSuspended);
        void __fastcall Prepare();
        void __fastcall ThreadPrepareSend::CalcSpeedGame();


};
//---------------------------------------------------------------------------
#endif
```

PrepareSend.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "PrepareSend.h"
#pragma package(smart_init)
#include "FormCtrl.h"
//---------------------------------------------------------------------------

//   Important: Methods and properties of objects in VCL can only be
//   used in a method called using Synchronize, for example:
//
//      Synchronize(UpdateCaption);
//
//   where UpdateCaption could look like:
//
//      void __fastcall Unit1::UpdateCaption()
//      {
//        Form1->Caption = "Updated in a thread";
//      }
//---------------------------------------------------------------------------

__fastcall ThreadPrepareSend::ThreadPrepareSend(bool CreateSuspended)
        : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall ThreadPrepareSend::Execute()
{
```

```
SpeedCalc = new SpeedCalculator;
SpeedCalc->ChangeTimeIntervalle(5);
Prepare();
}
//---------------------------------------------------------------------------
void __fastcall ThreadPrepareSend::Prepare()
{
PtrPaquet Paquet;
int passage = 0;

while(true)
  {
  Sleep(2);//temporaire , pour eviter saturation
  passage++;
  if (passage == 250) //tous les 250 passages
        {
        FormCtl->Speed = SpeedCalc->Speed();
        if (FormCtl->CheckBoxAutomateLocalBuffer->Checked)
           FormCtl->BufferLocal = BufferResizer(FormCtl->Speed,0,0,0,0,0);
        passage = 0;

        }

   // on attend après le recalcul de vitesse 50 passages , pour refaire un envoi de paquet
buffer
  if (passage == 50)
        {
        PtrPaquet Paquet;
        Paquet = new TPaquet;
        Paquet->PaquetType = 4; // 3 = Buffer  Size Change
        Paquet->Buffer = FormCtl->BufferLocal;
        FormCtl->ListOutputPrepareSend->AddPaquet(Paquet);
        //contrairement au autre paquet , autant le mettre ici.
        }
  // on va envoyer un paque pour signaler le changement de taille de buffer
  if (FormCtl->ListInputPrepareSend->Count() != 0)
    {
    Paquet = FormCtl->ListInputPrepareSend->Pos(0);
    if (Paquet->PaquetType == 1)
        {

          if (Paquet->Message == 144) // si c'est un note on
            {
                  FormCtl->StatusVectToSend->Notes[Paquet->NumNote] = true;
                  SpeedCalc->NotifyOn(true);
            }
          else
             if (Paquet->Message == 128)
                {
                FormCtl->StatusVectToSend->Notes[Paquet->NumNote] = false;
                }
        };

    if (Paquet->PaquetType == 2) //state
        {

        };
```

```
    if (Paquet->PaquetType == 3) //chat

        {

        };
    FormCtl->ListInputPrepareSend->RemoveFirst();
    FormCtl->ListOutputPrepareSend->AddPaquet(Paquet);
    }

  }
}
```

UDP FORWARDER

UDPPrinc.h

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "UdpPrinc.h"


//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TFormPrinc *FormPrinc;
//---------------------------------------------------------------------------
__fastcall TFormPrinc::TFormPrinc(TComponent* Owner)
        : TForm(Owner)
{
NMUDP2->RemotePort = 5001;
NMUDP1->RemotePort = 5000 ;


NMUDP1->LocalPort = 4040   ;
NMUDP2->LocalPort = 4041    ;


NbrPackRec = 0;
MidiDelay = new MidiDelayAdder(false);

//Liste de manipulaton pour les retards

PaquetListOutput = new ThreadMidiList();
PaquetListInput = new ThreadMidiList();

}
//---------------------------------------------------------------------------


void __fastcall TFormPrinc::NMUDP1DataReceived(TComponent *Sender,
      int NumberBytes, AnsiString FromIP)
{
```

```
    if (NumberBytes > 5 ) //on va dire que ç'est une taille minimale
    {
        NbrPackRec++;
        PtrPaquet Paquet = new TPaquet;
        EditNbrBytesRecu->Text = IntToStr(NumberBytes);
        EditNbrPaquetRecu->Text = IntToStr(NbrPackRec);
        EditDataRecFromIp->Text = FromIP;


        NMUDP1->ReadBuffer(Paquet->packt,NumberBytes,NumberBytes);

        Paquet->NumConn = 0;
        Paquet->NbrBytes = NumberBytes;
        PaquetListInput->AddPaquet(Paquet);
    }

}
//---------------------------------------------------------------------------

void __fastcall TFormPrinc::ButtonChangeClick(TObject *Sender)
{
NMUDP1->LocalPort = StrToInt(EditPortRec->Text);
NMUDP1->RemotePort = StrToInt(EditPortSend->Text);
NMUDP2->LocalPort = StrToInt(EditPortRec2->Text);
NMUDP2->RemotePort = StrToInt(EditPortSend2->Text);

}
//---------------------------------------------------------------------------


void __fastcall TFormPrinc::SendDelayedPaquet()
{

int i ;
int NumberBytes;
PtrPaquet Paquet;

  {

      if (!(PaquetListOutput->Count() == 0)) //tant qu'il y a des paquet à traiter
      {
      Paquet = PaquetListOutput->Pos(0); // on prend le premier
      NumberBytes = Paquet->NbrBytes; // on regarde la taille du paquet

      if (Paquet->NumConn == 0 )
        NMUDP1->SendBuffer(Paquet->packt,NumberBytes,NumberBytes); // on envoye le
buffer
      else
        NMUDP2->SendBuffer(Paquet->packt,NumberBytes,NumberBytes); // on envoye le
buffer
      EditNbrPackSend->Text = IntToStr(StrToInt(EditNbrPackSend->Text) +1);
      PaquetListOutput->RemoveFirst(); // on vire le paquet
  }
}
}

//---------------------------------------------------------------------------
```

```
void __fastcall TFormPrinc::UpDownMaxJitterChanging(TObject *Sender,
      bool &AllowChange)
{
  EditMaxJitter10Sec->Text =   StrToInt(UpDownMaxJitter->Position);
}
//---------------------------------------------------------------------------

void __fastcall TFormPrinc::UpDownDelayChanging(TObject *Sender,
      bool &AllowChange)
{
  EditDelay->Text =   StrToInt(UpDownDelay->Position);
}
//---------------------------------------------------------------------------

void __fastcall TFormPrinc::UpDownPercentLostChanging(TObject *Sender,
bool &AllowChange)
{
EditPercentLost->Text = UpDownPercentLost->Position;
}

//---------------------------------------------------------------------------


void __fastcall TFormPrinc::ButtonSendClick(TObject *Sender)
{
SendDelayedPaquet();
}
//---------------------------------------------------------------------------


void __fastcall TFormPrinc::NMUDP2DataReceived(TComponent *Sender,
      int NumberBytes, AnsiString FromIP)
{

      if (NumberBytes > 5 ) //on va dire que ç'est une taille minimale
      {

              NbrPackRec++;
              PtrPaquet Paquet = new TPaquet;

              EditNbrBytesRecu->Text = IntToStr(NumberBytes);
              EditNbrPaquetRecu->Text = IntToStr(NbrPackRec);
              EditDataRecFromIp->Text = FromIP;
              NMUDP2->ReadBuffer(Paquet->packt,NumberBytes,NumberBytes);
              Paquet->NumConn = 1 ; //cela signale que c'est un paquet RTCP
              Paquet->NbrBytes = NumberBytes;
              PaquetListInput->AddPaquet(Paquet);
       }

}
//---------------------------------------------------------------------------
```

MIDI Delay.cpp

```
//---------------------------------------------------------------------------
```

```
#include <vcl.h>
#include <sysutils.hpp>
#include <dos.h>
#pragma hdrstop
#include <stdlib.h>
#include <stdio.h>

#include "MidiDelay.h"
#include "ThreadListMidi.h"
#include "UdpPrinc.h"


#pragma package(smart_init)
//---------------------------------------------------------------------------

//   Important: Methods and properties of objects in VCL can only be
//   used in a method called using Synchronize, for example:
//
//      Synchronize(UpdateCaption);
//
//   where UpdateCaption could look like:
//
//      void __fastcall Unit1::UpdateCaption()
//      {
//        Form1->Caption = "Updated in a thread";
//      }
//---------------------------------------------------------------------------

__fastcall MidiDelayAdder::MidiDelayAdder(bool CreateSuspended)
        : TThread(CreateSuspended)
{

}
//---------------------------------------------------------------------------
void __fastcall MidiDelayAdder::Execute()
{
DelayList = new ThreadMidiList();
compteTours = 0 ;
Process();
}

//---------------------------------------------------------------------------
void __fastcall MidiDelayAdder::Process()
{



randomize();
while (true)
  {
    Sleep(3); // evite trop de saturation du processeur . voir si on peut pas faire autrement
    gettime(&Now);   // la ligne suivante nous donne le temps en dizaine de msec.
    TimeInMsec10=(Now.ti_hour * 60 * 60 * 100) + (Now.ti_min * 60 * 100) + (Now.ti_sec *
100) + (Now.ti_hund);
    if (!(FormPrinc->PaquetListInput->Count() == 0))
      {       //on calcule de délai et la gigue
              MaxJitter = StrToInt(FormPrinc->EditMaxJitter10Sec->Text);
```

```
                if (MaxJitter != 0) // on ne peut pas faire un modulo 0
                  RandomDelayToAdd = (rand()%(MaxJitter ));
                else RandomDelayToAdd = 0 ;  // si gigue Max etait 0
            DelayToAdd = (StrToInt(FormPrinc->EditDelay->Text));

        //on prend le paquet
        Paquet = FormPrinc->PaquetListInput->Pos(0);

        Lost = (rand()%100 < StrToInt(FormPrinc->EditPercentLost->Text)); //on calcule une
proba de le perdre
        //uniformité de proba,  10% de chance qu'un nbr de 1 a 100 soit > 10

        if(!Lost) //si la note n'est pas perdu on la met en output ATTENTION APRES IL FAUT
RETIRER LE PAQUET
            {
             Paquet->Msec10 = TimeInMsec10 + DelayToAdd + RandomDelayToAdd;
             DelayList->AddPaquet(Paquet);
            }
        FormPrinc->PaquetListInput->RemoveFirst();

        }

     ToAnalyse = DelayList->Count(); // nombre de paquet en input
     Analysed = 0;

     while (Analysed < ToAnalyse)
        {
         Paquet=(DelayList->Pos(Analysed));

        // si paquet le doit etre traite on le met dans la liste d'output et on
        // decremente le nombre de paquet à analyser
        // si le paquet ne doit pas encore etre traite on passe au paquet suivant
         if (Paquet->Msec10 <= TimeInMsec10)
             {
              FormPrinc->PaquetListOutput->AddPaquet(Paquet);
              FormPrinc->SendDelayedPaquet();
              DelayList->Remove(Analysed);
              ToAnalyse--;   //

              }
         else Analysed++;
         }//fin du while

  }//fin du while principal

}
//---------------------------------------------------------------------------
```

II Index des figures et des tables

Table 1 : Comparaison MIDI et Audio..17
Table 2 : Les instructions MIDI et les bytes de données correspondants....................23
Table 3 : Temps d'attaque et de relâchement pour différents types d'instrument.......48
Table 4 : Niveau de perception en fonction du temps de réponse et du tempo 54
Table 5 : Perception de la gigue en fonction du tempo 94
Table 6 : Niveau de perception en fonction du temps de réponse et du tempo 101

Figure 1 : Le modèle Internet...12
Figure 2 : Le modèle Studio...13
Figure 3 : Le modèle serveur de sons 14
Figure 4 : Configuration simple, Maître - Esclave...18
Figure 5 : Chaînage MIDI...18
Figure 6 : Configuration MIDI avec ordinateur...19
Figure 7 : Structure des messages MIDI 23
Figure 8 : Retard constant...30
Figure 9 : Retard variable 30
Figure 10 : Inversion de messages 31
Figure 11 : Perte du message Note On 33
Figure 12 : Perte du message Note Off 34
Figure 13 : Perte du message Note On et du message Note Off correspondant 35
Figure 14 : Ambiguïté lors de l'inversion de messages 36
Figure 15 : Ambiguïté lors de la perte de messages...37
Figure 16 : Intervalle de transmission 40
Figure 17 : Enveloppe ADSR 45
Figure 18 : Retard variable 46
Figure 19 : ADSR avec relâchement long 47
Figure 20 : ADSR avec relâchement court..48
Figure 21 : Transmission avec temps de réponse nul au niveau du clavier.................52
Figure 22 : Modification du temps de réponse 54
Figure 23 : Adaptation du temps de réponse 54
Figure 24 : Pile protocolaire mLAN...57
Figure 25 : Réseau mLAN 58
Figure 26 : Connectivité de réseau mLAN 58
Figure 27 : Pile protocolaire TCP/IP...60
Figure 28 : Format de l'en-tête du datagramme IP 62
Figure 29 : Format de l'entête TCP...65
Figure 30 : Format de l'entête UDP...65
Figure 31: Format du paquet RTP...68
Figure 32 : Utilisation de modèles pour la mesure de qualité sonore 86
Figure 33 : Interaction avec le matériel MIDI 89
Figure 34 : Mise en place du module de simulation 91

Figure 35 : Simulation du délai..92
Figure 36 : Intégration des fonctionnalités RTP 95
Figure 37 : Localisation de l'action du relais UDP..96
Figure 38 : Intégration du relais UDP 97
Figure 39 : Intégration des modules de gestion d'envoi et de réception.....................98
Figure 40 : Linéarisation du rapport Temps de réponse – Vitesse de jeu 102
Figure 41 : Schéma complet de l'application NetMidi 103

III Liste des acronymes utilisés

A

ADSR	Attack, Decay, Sustain, Release
ATM	Asynchronous Transfer Mode

B

BPM	Beats Per Minute

C

CBQ	Custom Based Queuing
CL	Controlled Load
CLNP	Connectionless Network Protocol
CNAME	Canonical Name
CNRS	Centre National de la Recherche Scientifique
cRTP	compressed Real-Time Protocol
CSRC	Contributing Source identifier

D

DNS	Domain Name System

F

FIFO	First In, First Out

G

GS	Guaranteed Service

I

IETF	Internet Engineering Task Force
IP	Internet Protocol
IPX	Internetwork Packet Exchange
ISDN	Integrated Services Digital Network

M

MCU	Multipoint Control Unit
MIDI	Musical Instrument Digital Interface
mLAN	music Local Area Network
MP3	Moving Picture Experts Group Layer-3 Audio (audio file format/extension)

O

OS	Operating System
OSI	Open Systems Interconnection

P

PCM	Pulse Code Modulation
PEAQ	Perceptual Evaluation of Audio Quality
POTS	Plain Old Telephone System
PSTN	Public Switched Telephone Network

Q

QoS	Quality of Service

R

RFC	Request for Comment
RSVP	Resource Reservation Protocol
RTCP	Real-Time Control Protocol
RTP	Real-Time Protocol

S

SIP	Session Initiation Protocol
SSRC	Synchronization Source identifier

T

TCP	Transmission Control Protocol

U

UDP	User Datagram Protocol
UIT	Union Internationale des Télécommunications
UMTS	Universal Mobile Telecommunications System

V

VoIP	Voice over Internet Protocol

IV Glossaire

A

ADSR (Attack, Decay, Sustain, Release) - Initiales des quatre segments dont sont constitués le ou les générateurs d'enveloppes d'un synthétiseur analogique, destinés à faire évoluer dans le temps l'un des paramètres d'un son (en premier lieu l'amplitude, mais aussi le timbre, la hauteur).

ATM (Asynchronous Transfer Mode) - Mode de transfert de données, qui permet d'acheminer à haut débit des paquets dont la principale caractéristique est de présenter une taille fixe. Le transfert est asynchrone, parce que les paquets de données, appelés cellules, de longueur fixe (48 octets pour le contenu et 5 pour l'adressage), n'occupent pas de place précise dans le temps, car ils sont acheminés à travers des voies différentes.

D

DNS (Domain Name System) - Système distribué de bases de données et de serveurs qui assure la traduction des noms de domaine utilisés par les internautes en numéros Internet utilisables par les ordinateurs, ceci pour permettre la transmission des messages d'un site à l'autre du réseau.

F

FIFO (First In, First Out) - Mode de traitement des données selon lequel les données entrées en premier seront les premières à être lues.

I

IETF (Internet Engineering Task Force) - Groupe informel et autonome, engagé dans le développement des spécifications pour les nouveaux standards d'Internet, et composé de personnes qui contribuent au développement technique et à l'évolution d'Internet et de ses technologies.

IP (Internet Protocol) - Protocole de base du réseau Internet qui régit l'expédition et la circulation des paquets de données à travers des réseaux hétérogènes.

IPX (Internetwork Packet Exchange) - Protocole utilisé pour acheminer les messages d'un noeud à un autre.

ISDN (Integrated Services Digital Network) - Réseau de transmission entièrement numérique, qui est capable de fournir ou de supporter une vaste gamme de services de télécommunication.
La principale caractéristique d'un réseau numérique à intégration de services est de permettre l'agrégation de canaux qui présentent chacun un débit de 64 kilobits par seconde. Ainsi, un accès de base qui comporte deux canaux permet d'atteindre 128

kilobits par seconde. Un réseau numérique à intégration de services permet d'échanger des sons, des données ou des images, de telle manière qu'on peut l'utiliser pour offrir des services comme la téléphonie, la visiophonie, la télécopie, la messagerie électronique, etc.

M

MCU (Multipoint Control Unit) - Dans un système de communication utilisant la norme H.323, unité qui assure la gestion des conférences comportant trois participants ou plus.

MIDI (Musical Instrument Digital Interface) - Interface permettant d'interconnecter des instruments de musique numériques entre eux (synthétiseurs, séquenceurs, boîte à rythmes, etc.) ou avec un micro-ordinateur, grâce à laquelle ils peuvent échanger des données.

Multicast - Technique permettant d'adresser un paquet d'information à un sous-ensemble d'un réseau. Cette technique d'échange de l'information est aussi appelée point à multipoint. Généralement le transfert d'information dans un réseau se fait:
- soit point à point en utilisant comme source et destination des adresses unicast,
- soit en faisant de la diffusion(broadcast).

O

OS (Operating System) - Logiciel de base d'un ordinateur destiné à commander l'exécution des programmes en assurant la gestion des travaux, les opérations d'entrée/sortie sur les périphériques, l'affectation des ressources aux différents processus, l'accès aux bibliothèques de programmes et aux fichiers ainsi que la comptabilité des travaux (par exemple, MS-DOS, OS/2, Windows, UNIX, etc.).

OSI (Open Systems Interconnection) - Modèle de référence défini par ISO constitué de 7 couches, chacune spécifiant les fonctions et les protocoles nécessaires à deux nœuds pour communiquer en utilisant l'infrastructure réseau sous-jacente (médium physique, commutateurs, routeurs, ponts, multiplexeurs, nœuds intermédiaires). Ce cadre conceptuel défini par l'ISO (International Standard Organisation) permet de normaliser l'échange entre réseaux hétérogènes.

P

PCM (Pulse Code Modulation) - Processus selon lequel un signal analogique est converti en un signal numérique par échantillonnage, quantification et codage.

POTS (Plain Old Telephone System) - Service téléphonique traditionnel de base fonctionnant en mode analogique sur des lignes individuelles en fil de cuivre donnant accès au réseau téléphonique public commuté.

PSTN (Public Switched Telephone Network) - Réseau téléphonique fixe traditionnel dont le fonctionnement est basé sur la commutation de circuits.

Q

QoS (Quality of Service) - Aptitude d'un service à répondre adéquatement à des exigences, exprimées ou implicites, qui visent à satisfaire ses usagers.
Ces exigences peuvent être liées à plusieurs aspects d'un service : son accessibilité, sa continuabilité, sa disponibilité, sa fiabilité, sa maintenabilité, etc. Dans le domaine des télécommunications, il est aussi utilisé, dans un sens spécifique, comme synonyme de qualité d'écoulement du trafic.

R

RFC (Request for Comment)- Documents qui constituent les normes pour le réseau Internet et les protocoles de la famille TCP/IP. L'élaboration des RFCs comporte plusieurs étapes placées sous le contrôle de l'IETF.

Routeur - Désigne un équipement qui assure la fonction d'acheminement (routage) d'une communication à travers un réseau (niveau 3 du modèle OSI).

RSVP (Resource Reservation Protocol) - Protocole de réservation de ressources de bout en bout permettant entre autre la négociation d'une Qualité de Service.

RTCP (Real-Time Control Protocol) - Protocole qui permet de gérer les communications entre les participants à une conférence multipoint et qui vient s'ajouter au protocole de transmission en temps réel que l'on désigne habituellement par le sigle «RTP».

RTP (Real-Time Protocol) - Protocole de transmission en temps réel de la voix et des images sur des réseaux à commutation de paquets de type TCP-IP, comme des Intranets ou Internet.

T

TCP (Transmission Control Protocol) - Protocole Internet responsable de la préparation des données sous forme de paquets avant l'expédition et de leur assemblage dans le bon ordre à la réception.
Le protocole TCP correspond à la couche de transport du modèle OSI.

U

UDP (User Datagram Protocol) - Protocole utilisant le protocole IP pour expédier des datagrammes d'une application Internet à l'autre.
Considéré comme peu fiable, le protocole UDP est utilisé pour la transmission de petits messages où, a priori, il n'y a pas de problème de séquencement.
On dit du protocole UDP qu'il est « sans connexion » parce qu'à la différence du protocole TCP, il n'exige pas de l'expéditeur et du récepteur d'établir une connexion avant la transmission des datagrammes.

UIT (Union Internationale des Télécommunications) - Institution spécialisée des Nations Unies crée en 1865 en vue notamment de réglementer les services télégraphiques, téléphoniques et radio-électriques.

UMTS (Universal Mobile Telecommunications System) - Système de téléphonie cellulaire de troisième génération, de conception européenne, qui vise à permettre aux appareils de supporter des débits pouvant aller jusqu'à 2 mégabits par seconde (Mbit/s), de manière à rendre plus facile l'accès à Internet et à faire en sorte que les communications multimédia puissent s'effectuer rapidement.

V

VoIP (Voice over Internet Protocol) - Technique permettant d'intégrer la voix aux données transmises par paquets sur un réseau utilisant le protocole Internet.