Une approche du patching audio collaboratif

Eliott Paris

UNIVERSITÉ PARIS VIII
VINCENNES – SAINT-DENIS
UFR 1 – ARTS, PHILOSOPHIE, ESTHÉTIQUE

THÈSE
Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PARIS VIII
En
ESTHÉTIQUE, SCIENCES ET TECHNOLOGIES DES ARTS

Discipline : Musique
Présentée et soutenue publiquement le 5 décembre 2018 par

Eliott PARIS

Titre :

UNE APPROCHE DU PATCHING AUDIO COLLABORATIF : ENJEUX ET DEVELOPPEMENT DU COLLECTICIEL KIWI

***Thèse dirigée par* Anne SÈDES *et* Alain BONARDI**

Jury :

Gérard ASSAYAG (Rapporteur)
Pierre COUPRIE (Rapporteur)
Myriam DESAINTE-CATHERINE (Présidente)
Anne SÈDES (Directrice)
Alain BONARDI (Co-directeur)

À Maurice et Jean

RESUME

Les logiciels de patching audio traditionnels, tels que Max ou Pure Data, sont des environnements qui permettent de concevoir et d'exécuter des traitements sonores en temps réel. Ces logiciels sont mono-utilisateurs ; or, dans bien des cas, les utilisateurs ont besoin de travailler en étroite collaboration à l'élaboration ou à l'exécution d'un même traitement. C'est notamment le cas dans un contexte pédagogique ainsi que pour la création musicale collective. Des solutions existent, mais ne conviennent pas forcément à tous les usages. Aussi avons-nous cherché à nous confronter de manière concrète à cette problématique en développant une nouvelle solution de patching audio collaborative, baptisée Kiwi, qui permet l'élaboration d'un même traitement sonore à plusieurs mains de manière distribuée. À travers une étude critique des solutions logicielles existantes nous donnons des clefs de compréhension pour appréhender la conception d'un système multi-utilisateur de ce type. Nous énonçons les principaux verrous que nous avons eu à lever pour rendre cette pratique viable et présentons la solution logicielle. Nous exposons les possibilités offertes par l'application et les choix de mise en œuvre techniques et ergonomiques que nous avons faits pour permettre à plusieurs personnes de coordonner leurs activités au sein d'un espace de travail mis en commun. Nous revenons ensuite sur différents cas d'utilisation de ce collecticiel dans un contexte pédagogique et de création musicale afin d'évaluer la solution proposée. Nous exposons enfin les développements plus récents et ouvrons sur les perspectives futures que cette application nous permet d'envisager.

Mots-clés : *Informatique musicale, patching audio, collecticiel, édition et jeu collaboratif en temps réel, interfaces multi-utilisateurs.*

ABSTRACT

Title: An approach of collaborative audio patching: Challenges and development of the *Kiwi* groupware.

Traditional audio patching software, such as Max or Pure Data, are environments that allow you to design and execute sound processing in real time. These programs are single-user, but, in many cases, users need to work together and in a tight way to create and play the same sound processing. This is particularly the case in a pedagogical context and for collective musical creation. Solutions exist, but are not necessarily suitable for all uses. We have tried to confront this problem in a concrete way by developing a new collaborative audio patching solution, named Kiwi, which allows the design of a sound processing with several hands in a distributed manner. Through a critical study of the existing software solutions we give keys of comprehension to apprehend the design of a multi-user system of this type. We present the main barriers that we had to lift to make this practice viable and present the software solution. We show the possibilities offered by the application and the technical and ergonomic implementation choices that we have made to allow several people to coordinate their activities within a shared workspace. Then, we study several uses of this groupware in pedagogical and musical creation contexts in order to evaluate the proposed solution. Finally, we present the recent developments and open up new perspectives for the application.

Keywords: *Computer music, audio patching, groupware, real-time collaborative editing and playing, multi-user interfaces*.

REMERCIEMENTS

Je tiens à remercier chaleureusement mes deux directeurs de recherche, Anne Sèdes et Alain Bonardi, qui ont soutenu et accompagné mon parcours depuis maintenant plus de sept ans, je vous dois à tous les deux énormément !

Je remercie aussi les membres du jury d'avoir accepté d'évaluer ce travail de thèse.

Je remercie aussi évidemment le Labex Arts-H2H, de m'avoir accordé un contrat doctoral, et l'ANR d'avoir financé le projet MUSICOLL, cela nous a offert un cadre idéal de travail pour mener à bien cette recherche.

Merci à toute l'équipe du projet MUSICOLL, l'équipe *OhmForce* tout d'abord – F. Bacquet, R. Dinge, G. Makles, V. Bireben et L. De Sauras – pour leur enthousiasme, leur expérience et leurs précieux conseils, avec qui nous avons collaboré, en temps réel ou différé, en amont et durant tout le projet. A. Bonardi et A. Sèdes pour leur engagement et leur encadrement, P. Guillot et J. Millot, avec qui j'ai partagé cette aventure de recherche et de développement tout au long du projet. À E. Maestri qui a eu le courage de donner un cours à l'Université durant tout un semestre avec un logiciel encore en phase de développement, et à P. Galleron, pour avoir notamment contribué à l'élaboration de ce cours.

Merci à mes amis, en particulier à Pierre, avec qui j'ai partagé tant de discussions palpitantes sur le présent sujet, qui m'a toujours soutenu et accompagné ! Merci aussi à Pauline !

Merci à ma famille, mes parents, ma sœur, Emma, qui a composé avec mes humeurs durant toute cette aventure, à mon grand-père de m'avoir promis un déjeuner dans un grand restaurant si j'obtenais un jour mon doctorat, il y a de ça plus de vingt ans… Je ne suis pas sûr qu'il s'en souvienne, moi si ;).

Merci aussi à tous les Arnould, qui m'ont supporté et soutenu durant ces derniers mois de rédaction ! Enfin merci à Ségolène, qui a partagé mes doutes, mes angoisses, m'a réconforté et toujours encouragé à aller jusqu'au bout, « Allez ! », cela semble avoir fonctionné ! Encore merci ma Sé. !

SOMMAIRE

AVANT-PROPOS

Ce travail de thèse a été financé par un contrat doctoral de trois ans, accordé par le Labex Arts-H2H de l'Université Paris 8[1]. Il été en partie à l'origine du projet ANR-MUSICOLL[2], qui a associé le CICM[3], sous équipe d'accueil EA 1572 MUSIDANSE et la PME *OhmForce*[4] sur une période de trois ans, de janvier 2016 à décembre 2018.

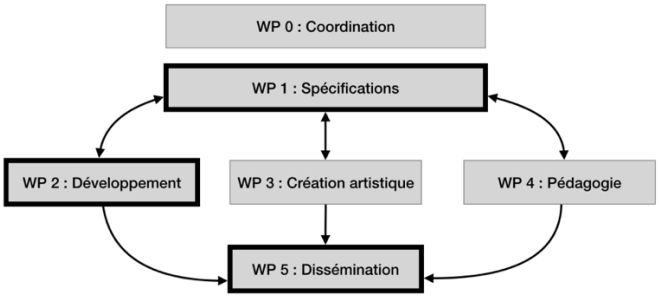

Figure 1 – Schéma présentant l'organisation du projet MUSICOLL, répartie en différents *Work Package* (*WP*) tels que soumis à l'ANR lors de la proposition initiale. Les rectangles de bordures plus épaisses symbolisent les WP dans lesquels nous avons été le plus impliqué au sein du projet et où s'est inscrit plus spécifiquement le travail présenté dans le cadre de ce mémoire de thèse.

Les principaux résultats attendus de ce projet étaient : d'une part, la production d'une maquette d'environnement collaboratif multiplateforme de création musicale, nommée Kiwi[5], qui permette de réaliser des modules de synthèse et de traitement sonore à plusieurs mains en temps

[1] http://www.labex-arts-h2h.fr/ [consulté le 05/09/18].

[2] Musique temps réel collaborative et nomade, n°ANR-15-CE38-0006-01 : http://musicoll.mshparisnord.org/ [consulté le 05/09/18].

[3] Centre de Recherche en Informatique et Création Musicale : http://cicm.mshparisnord.org/ [consulté le 05/09/18].

[4] https://www.ohmforce.com/ [consulté le 05/09/18].

[5] L'ensemble du code source ainsi que les différents liens de téléchargement de l'application sont disponibles à partir de l'adresse suivante : https://github.com/Musicoll/Kiwi [consulté le 05/09/18].

réel ; et d'autre part, l'évaluation de cette solution logicielle sur le plan de la création musicale et du renouvellement de l'approche pédagogique traditionnelle de l'enseignement du patching audio[6].

Notre travail au sein du projet MUSICOLL s'est inscrit plus spécifiquement dans les lots 1, 2 et 5, dédiés respectivement aux spécifications, au développement logiciel et à la dissémination scientifique [Figure 1]. Le contenu de cette présente étude embrasse de manière rétrospective un travail collectif réalisé par plusieurs personnes au sein de l'équipe du projet MUSICOLL et ne saurait donc à ce titre être attribué uniquement à l'auteur de ce document. Aussi nous faut-il préciser la manière dont ont été répartis les différents travaux de recherche effectués dans ce cadre, ceux-ci étant bien évidemment interdépendants.

La tâche de coordination du projet a été assurée par Alain Bonardi du côté du CICM et Franck Bacquet du côté d'*OhmForce*. Les spécifications ont été réalisées en équipe, essentiellement par le CICM (Alain Bonardi, Anne Sèdes, Eliott Paris, Pierre Guillot, Jean Millot, Philippe Galleron, Eric Maestri), mais avec le soutien à la fois logistique, humain et technologique de l'équipe *OhmForce* (Franck Bacquet, Raphaël Dinge, Gregory Makles, Vincent Bireben et Laurent De Sauras).

Le développement logiciel de l'application Kiwi a principalement été assuré par trois personnes[7] : J. Millot, engagé en tant qu'ingénieur de recherche sur une période allant de février 2016 à avril 2018 ; P. Guillot, en tant que doctorant en amont, sur la première année du projet,

---

[6] Tel qu'il est connu, par exemple dans le domaine des arts multimédia et des pratiques musicales expérimentales, notamment dans des logiciels comme Max ou Pure Data.

[7] https://github.com/Musicoll/Kiwi/graphs/contributors et https://github.com/Musicoll/kiwi-node-server/graphs/contributors [consultés le 13/09/18].

et sur la fin en qualité de post-doctorant d'avril à août 2018 ; et enfin l'auteur de ce document, en amont puis sur l'ensemble du projet. Ce développement logiciel n'aurait pas pu se faire non-plus sans le soutien de l'équipe *OhmForce* qui nous a apporté son expertise, de nombreux conseils techniques et notamment sa technologie *flip*[8] qui en constitue le noyau central.

Les lots dédiés à la création musicale et à la pédagogie sur le projet ont été assurés une fois encore par l'ensemble de l'équipe. Nous noterons néanmoins que la conception du nouveau cours pilote réalisé avec Kiwi comme support logiciel à l'Université Paris VIII a fait intervenir plus spécifiquement P. Galleron, docteur, enseignant et pédagogue engagé en qualité de post-doctorant sur le projet et E. Maestri, docteur, et compositeur, en qualité d'A.T.E.R.

Le lot dédié à la dissémination visait l'exposition des résultats du projet dans son ensemble, celui de la recherche, du développement, de la pédagogie et de la création sonore. L'équipe a dans ce cadre pu effectuer plusieurs présentations et articles auxquels l'auteur a contribué [Paris & al., 2017], [Sèdes & al., 2017, 2018], [Galleron & al., 2018]. Ce mémoire de thèse s'inscrit aussi dans cette même perspective.

[8] La bibliothèque *flip* est développée et commercialisée par la société *irisate*, fondée par plusieurs membres de l'équipe *OhmForce*. https://irisate.com/ [consulté le 05/09/18].

INTRODUCTION GENERALE

Depuis les débuts de la musique électronique, les outils de création mis à la disposition des compositeurs n'ont eu de cesse d'évoluer, d'influencer leurs relations sociales, leur organisation en communauté voire même leur inscription dans des mouvements esthétiques [Laliberté, 2013]. Nous pouvons en première approche distinguer trois configurations socio-technologiques dans l'histoire de la musique électronique, qui se sont adaptées progressivement aux trois ères majeures de l'informatique. La première, celle de l'ordinateur centralisé, peut être rapprochée, en musique, de l'âge des studios institutionnels, de 1950 aux années 1990. L'équipement informatique nécessaire à la création musicale électronique n'était alors disponible que dans ces studios et les compositeurs devaient nécessairement passer par ces institutions ; cela a permis à des communautés de pratique de se constituer et une mise en commun des savoir-faire de s'opérer. La deuxième, celle de l'ordinateur personnel, correspond à la démocratisation des outils musicaux et à l'avènement du *home studio*, entre les années 1990 et 2000. Les compositeurs ont alors pu s'approprier de nouveaux outils de création et effectuer l'essentiel de leur travail à domicile sur leur machine, mais cela a pu mener aussi à un isolement des pratiques qui y sont associées. La dernière, celle où nous évoluons actuellement, est celle des interfaces mobiles et connectées, des services en ligne et du collaboratif. Elle est marquée par la mise en réseau des créateurs, des produits de leur création et enfin de leurs outils. Dans ce contexte, de nouveaux logiciels musicaux émergent tels que le séquenceur *OhmStudio*[9], créé pour pouvoir faire du montage audio à plusieurs mains et à distance via internet. Les utilisateurs se retrouvent virtuellement au sein d'un même espace de création musicale, ils peuvent observer les autres en train de travailler en même temps qu'eux, communiquer plus facilement et ainsi apprendre de leurs pratiques respectives.

[9] http://www.ohmstudio.com/ [consulté le 05/09/18].

Cette thèse s'intéresse aux pratiques collaboratives dans le domaine plus spécifique de la création musicale associée au temps réel et aux moyens techniques qui permettent de les soutenir. Les outils les plus représentatifs dans le domaine du temps réel sont actuellement les logiciels Max[10] [Zicarelli, 2002] et Pure Data[11] [Puckette, 1988, 2002, 2009]. Ces logiciels sont des environnements de patching audio modulaires qui permettent, entre autres, de concevoir et d'exécuter des traitements sonores. Ils sont utilisés par une vaste communauté de chercheurs, de musiciens, d'enseignants, de programmeurs ou d'artistes multimédia qui les utilisent notamment pour prototyper de nouveaux traitements sonores, former les élèves au traitement du signal audio, créer des pièces musicales mixtes, des installations ou encore exécuter des performances électroniques en *live*. Ces environnements sont mono-utilisateurs. Aussi, la pratique du patching audio se fait donc essentiellement seul sur un ordinateur individuel. Cependant, dans de nombreux cas, les utilisateurs cherchent à mettre en commun leurs traitements sonores, leurs patchs, pour pouvoir les consulter, y apporter des modifications ou encore les utiliser à plusieurs. Quelles sont les moyens traditionnels qui permettent le partage de ces données ? L'une des réponses actuelles au partage de patch est en fait la même que celle apportée au partage de tout document numérique. Ils sont le plus souvent envoyés par courrier électronique, ou partagés comme fichiers sur des espaces de stockage en ligne. Des utilisateurs s'emparent, par exemple, d'un forum de discussion en ligne pour y stocker un patch et l'éditer à plusieurs à tour de rôle [Digiology, 2010a]. Plusieurs personnes peuvent donc accéder à ces patchs pour les éditer ou les exécuter localement sur leur machine. Mais cela conduit le plus souvent à une gestion manuelle des conflits entre versions. Généralement, sans outils

[10] Solution propriétaire développée et commercialisée par la société Cycling'74 : https://cycling74.com/ [consulté le 22/09/18].

[11] Solution *open-source* développée et maintenue par Miller Puckette et d'autres contributeurs : https://puredata.info/ [consulté le 22/09/18].

collaboratifs adaptés, le dialogue inter-utilisateur passe par la mise à disposition de traces de l'activité telles que des copies d'écran ou des annotations périphériques au document de travail qu'un des créateurs diffuse aux autres. D'autres pratiques collaboratives sont néanmoins déjà bien installées. Des utilisateurs réunis au même endroit peuvent contrôler l'exécution d'un même traitement en temps réel. D'autres, dispersés géographiquement initient des *jam session* et jouent ensemble sur des instruments numériques programmés avec Pure Data à travers un réseau [Haefeli, 2013]. D'autres encore pratiquent le *live coding* en éditant et en exécutant le même patch côte à côte depuis plusieurs machines [Zmölnig, 2007], [Pd~Graz, 2009]. Si ces pratiques sont révélatrices d'un intérêt musical généré par la collaboration au sein de ces environnements, nous verrons que les moyens techniques qui viennent les soutenir ne sont pas toujours idéals et ne conviennent pas forcément à tous les usages. Aussi nous faut-il trouver de nouvelles solutions logicielles qui puissent répondre plus spécifiquement à ces enjeux.

Les logiciels multi-utilisateurs – appelés aussi collecticiels ou *groupware* en anglais – sont, contrairement aux logiciels mono-utilisateurs, conçus spécifiquement pour favoriser le travail collaboratif [Ellis & al., 1991], [Salber, 1995]. Le logiciel *OhmStudio* cité plus haut s'inscrit dans cette catégorie et plus particulièrement dans celle des logiciels permettant une édition collaborative de document en temps réel et de manière distribuée [Greenberg, 2002]. La suite d'outils en ligne *Google Docs*[12] en est un autre exemple. Elle permet notamment le stockage de documents en ligne et une édition collaborative de leur contenu. Les personnes peuvent accéder de n'importe quel endroit et à n'importe quel moment à ces documents, y contribuer en même temps sans que cela ne génère de conflits et interagir les uns avec les autres à travers un espace de travail virtuel commun.

[12] https://www.google.fr/intl/fr/docs/about/ [consulté le 05/09/18].

Le présent sujet de recherche s'intéresse précisément à la fusion qui pourrait être opérée entre ces deux types d'applications que sont d'une part les systèmes multi-utilisateurs et d'autre part les environnements de patching audio. Aussi, comment envisager la conception d'une solution de ce type ? Le développement d'une solution collaborative de patching audio pose de nombreuses questions. L'une d'entre-elles sera de savoir ce qui doit réellement être partagé entre les utilisateurs au sein du logiciel. Mais aussi, comment permettre à plusieurs personnes d'éditer le même patch sans pour autant générer de conflit [Ellis & Gibbs, 1989] ? Ou encore, comment permettre aux utilisateurs de collaborer à distance en temps réel [Greenberg, 2002] ? Dans un contexte où ils ne peuvent plus se voir directement, comment ont-ils conscience du travail réalisé par les autres et comment représenter l'action du groupe au sein de l'espace de travail partagé [Gutwin, 1997], [Gutwin & Greenberg, 1997] ?

À travers l'étude des pratiques collaboratives existantes, nous tenterons de mettre en lumière ces différentes problématiques. Nous proposerons ensuite des réponses concrètes en présentant le collecticiel Kiwi, développé dans le cadre du projet MUSICOLL.

Cette présentation s'échelonne sur huit chapitres organisés au sein de trois parties. La première partie a pour but dégager les principaux enjeux à la fois musicaux, pédagogiques, scientifiques et techniques liés à la collaboration en général et plus précisément lorsqu'elle s'applique aux activités pratiquées à travers les logiciels de patching audio traditionnels. Dans le premier chapitre, nous donnerons une vision plus précise du sujet. Nous définirons mieux les caractéristiques des environnements de patching audio traditionnels. Nous distinguerons alors les deux types d'activités qui peuvent s'y pratiquer, à savoir celle de la conception de patch et celle relative à leur exécution. Nous aborderons ensuite les aspects relatifs à la collaboration au

sens large, aux différents types d'interaction, mais aussi plus précisément aux fonctions proposées par les logiciels collaboratifs. Pour cela, nous donnerons des premiers jalons conceptuels qui nous aiderons à analyser et classer les systèmes multi-utilisateurs. Cela nous permettra de dégager les différentes activités, interactions, et fonctions attendues ou souhaitées au sein d'une solution de patching audio utilisable de manière collaborative. Dans le second chapitre, nous étudierons en détail et de manière critique les pratiques existantes et analyserons les solutions logicielles mises en place pour les soutenir. Cette série d'études fera ressortir la dimension ludique, pédagogique et musicale de la pratique, mais aussi des problématiques plus techniques liées à la conception de ce genre de système. Nous les formaliserons dans le troisième chapitre afin de définir les enjeux plus spécifiques de la solution logicielle Kiwi que nous avons créée.

Dans la seconde partie, nous présenterons Kiwi, une nouvelle application de patching qui se situe dans la lignée des logiciels traditionnels de type Max ou Pure Data, mais qui permet l'élaboration collaborative de traitements sonores en temps réel. Nous nous appuierons pour cela sur les outils conceptuels proposés dans la première partie de ce mémoire. Dans le quatrième chapitre, nous présenterons les spécificités graphiques et fonctionnelles du langage, l'architecture logicielle de la solution et l'interface permettant l'élaboration et l'exécution de traitements sonores. Nous spécifierons aussi la nature des données partagées entre les utilisateurs de ce qui relève de leur espace privé et local. Nous étudierons ensuite dans le cinquième chapitre les aspects liés à la coordination des activités de groupe au sein de l'espace partagé et expliciterons les différents choix de mise en œuvre qui ont été effectués pour y parvenir. Nous terminerons cette présentation du logiciel dans le sixième chapitre en abordant l'aspect communicationnel de l'application en exposant une première solution permettant aux

utilisateurs de partager de l'information en temps réel pouvant ouvrir la voie à des pratiques musicales de jeu collaboratif.

Le projet MUSICOLL n'est pas arrivé à son terme au moment de la rédaction de ce document et l'application que nous présentons dans ce cadre est toujours en développement. Il s'agira alors dans la troisième partie de ce mémoire d'évaluer le travail réalisé jusqu'ici en confrontant les possibilités qu'elle offre dans son état actuel avec les enjeux exposés en première partie. Le septième chapitre sera donc consacré à cette évaluation en nous appuyant notamment sur les différents projets réalisés jusqu'ici avec Kiwi. Cela nous permettra de déterminer le type de pratique auquel permet de répondre l'application, son potentiel musical et pédagogique, mais aussi certaines de ses limites actuelles. Nous ouvrirons ensuite le sujet en nous projetant vers l'avenir de cette solution logicielle. Le huitième chapitre exposera les développements récents et donnera des perspectives de recherche et de développement à plus long terme visant à l'amélioration de cette application.

Partie I – Présentation, étude et formalisation des enjeux

Le but de cette première partie est de définir les enjeux liés à la collaboration au sein des environnements de patching audio traditionnels et à la conception d'un système répondant à ces enjeux. Dans le premier chapitre, nous donnerons un aperçu général du sujet qui nous aidera à mieux définir, d'une part, ce qu'est un patch, mais aussi les différents aspects relatifs à la notion de collaboration. Dans le second chapitre, nous étudierons de manière critique différents outils et usages qui sont pour nous représentatifs du paysage actuel dans ce domaine. Cela nous permettra de faire émerger les principales attentes de la part des utilisateurs mais aussi des problématiques, notamment techniques, liées à la conception de ces outils. Nous reviendrons ensuite dans le dernier chapitre sur les différentes notions abordées pour y apporter des précisions en nous appuyant notamment sur les différentes solutions logicielles étudiées. Nous dégagerons alors ce que nous considérons comme les principaux enjeux et les verrous à lever pour rendre la pratique du patching audio collaborative viable en envisageant la solution idéale d'un point de vue technique qui permettrait de la soutenir.

1. Aperçu général du sujet

Pour donner un aperçu général du sujet, il nous faut en premier lieu mieux définir l'objet sur lequel se porte l'étude de la collaboration et des solutions collaboratives au sein de cette recherche, à savoir le patch. Nous préciserons dans un second temps ce que nous entendons par collaboration et donnerons notamment des premiers jalons conceptuels qui nous permettront d'analyser les types d'interaction et les fonctions que sont à même de supporter les systèmes logiciels multi-utilisateurs. Nous pourrons alors imaginer les enjeux en matière de pratique liés à la collaboration au sein des environnements de patching.

1.1. Patching

La notion de patching est relative aux activités réalisées et supportées au sein des environnements de patching. Après avoir donné une définition générale du patch, nous distinguerons les deux activités principales permises par ce type d'environnement qui sont d'une part l'activité de conception et d'autre part l'activité de jeu.

1.1.1. Définition

Un patch peut être défini comme un programme informatique, réalisé au sein d'un environnement de programmation graphique spécifique [Myers, 1990]. La programmation d'un patch se fait grâce à un assemblage de briques essentielles, appelées aussi objets, reliées entre elles par des liens, pour créer des traitements musicaux, ou plus généralement des processus interactifs qui peuvent être exécutés et contrôlés en temps réel au sein du même environnement. On parle d'environnement pour désigner ce genre de logiciels dans la mesure où ils se comportent à la fois comme des éditeurs, des environnements de développement donc, et des environnements d'exécution pour les programmes créés à travers eux.

Il existe de nombreux logiciels permettant de faire du patching audio[13], mais les deux plus connus et utilisés aujourd'hui, et auxquels nous nous réfèrerons plus principalement dans le cadre de notre recherche, sont Max et Pure Data[14]. Le logiciel Max est un logiciel propriétaire commercialisé par la société Cycling'74 depuis 1997. Le logiciel Pure Data est un logiciel libre, développé depuis 1996 par Miller Puckette et d'autres contributeurs [Puckette, 1997]. Ces deux

[13] On pourra citer comme exemple le logiciel *Usine* (http://www.sensomusic.org/), *Bidule* (https://www.plogue.com/produits/bidule), ou encore *AudioMulch* (http://www.audiomulch.com/). [liens consultés le 05/09/18].

[14] La version des logiciels à laquelle nous nous référons dans le cadre de cette thèse correspond dans le cas de Max à la version *7.3.5*. et dans le cas de Pure Data à la version *0.48-1*.

logiciels proviennent d'un paradigme de programmation commun, surnommé « Max » [Puckette, 2002], et développé initialement à l'IRCAM à la fin des années 1980 par Miller Puckette [Puckette, 1988]. Par extension, les environnements mettant en œuvre ce paradigme sont appelés des environnements « Max-like ». Dans la mesure où le logiciel commercial « Max/MSP » a été renommé simplement « Max » dans les récentes versions, il nous paraît désormais ambigüe de se référer à ce nom pour parler du paradigme général sous-jacent, aussi nous préfèrerons employer le terme « langage Patcher » pour le désigner, ce qui nous parait être un terme plus générique signifiant la même chose.

Un patch peut être vu graphiquement comme un espace d'abord vierge, sur lequel l'utilisateur vient construire des graphes qui modélisent des processus. Les nœuds de ces graphes sont représentés au sein du patch sous la forme de boîtes textuelles ou graphiques qui permettent de spécifier le fonctionnement interne de celui-ci, à partir d'un nom et d'une liste d'arguments (optionnels ou non suivant les cas). Selon le type d'objet créé, celui-ci pourra avoir une apparence, un comportement et un nombre d'entrées et de sorties différent. Les entrées et sorties des objets, qui sont appelées respectivement *inlets* et *outlets*, peuvent être connectées entre-elles grâce à des liens qui modélisent une communication unidirectionnelle d'un objet vers un autre au sein du graphe. Si un rapprochement pouvait être fait avec les langages de programmation textuels plus traditionnels, on pourrait dire que les entrées des objets représentent les arguments ou paramètres d'une fonction, tandis que les sorties correspondent au résultat de cette fonction. L'une des spécificités du langage Patcher est que deux types de données peuvent transiter par les objets au sein du patch lors de son exécution, appelées

communément données de contrôle et de signal[15]. Les données de contrôle sont des événements ponctuels, des messages généralement apériodiques, mais pas nécessairement, produits à une fréquence relativement faible. Le signal représente quant à lui une suite d'échantillons audio calculés à une fréquence élevée, fixée par la fréquence d'échantillonnage du système.

1.1.2. Activités de conception et de jeu

On distingue deux activités principales au sein de ces environnements :

- L'activité de conception du traitement sonore, qui se comprend donc comme l'édition graphique du programme.
- L'activité de jeu qui permet la pratique musicale du *live* par le contrôle du programme lors de son exécution.

L'activité d'édition du patch consiste à modéliser le processus ou le traitement en question au sein de l'interface du logiciel. Les opérations réalisables lors de cette activité sont par exemple l'ajout ou la suppression d'objet, leur modification, leur positionnement dans l'espace graphique du patch ou encore leur raccordement. L'utilisateur qui modélise un traitement dans un environnement de patching définit aussi les entrées et les sorties du programme. Ces entrées et sorties peuvent prendre la forme d'un signal audio reçu depuis ou envoyé vers la carte son, de données issues de capteurs, de données transitant sur un réseau etc. Les entrées et sorties du programme peuvent aussi être intégrées au patch sous la forme de boîtes-objets spécifiques appelées objets de contrôle ou encore objets graphiques. Dans ce cas, l'utilisateur peut agir directement sur le patch durant son exécution en interagissant avec ces objets.

[15] Notons que d'autres types de données peuvent coexister au sein des environnements de patching traditionnels tels que Max ou Pure Data, permettant notamment la mise en place de traitements vidéos. Ils sont ignorés dans le cadre de cette étude.

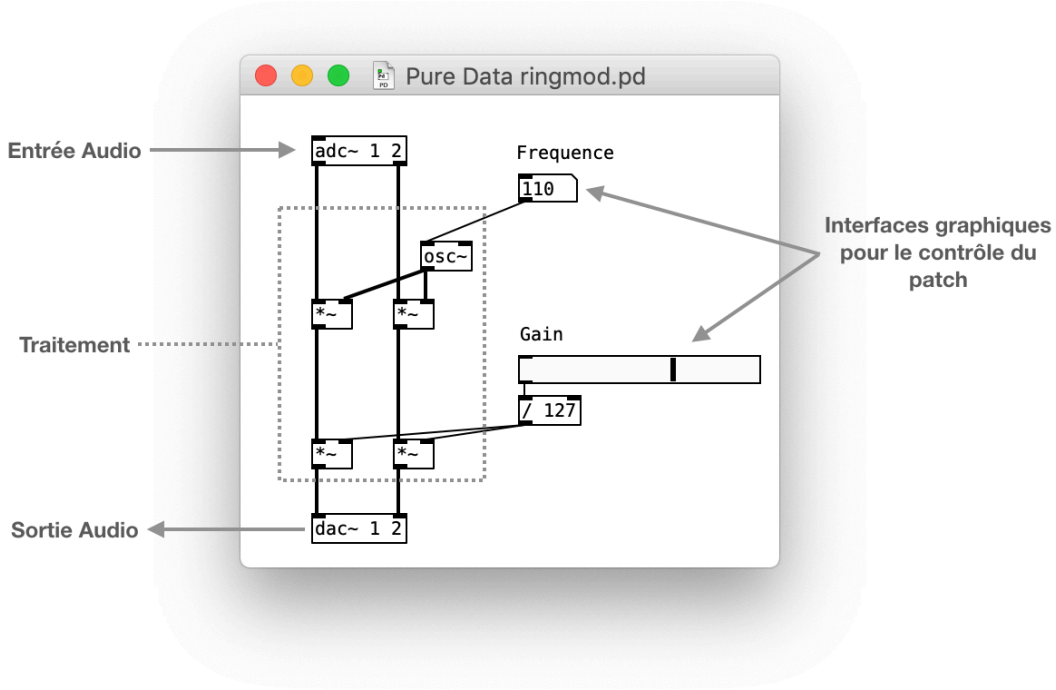

Figure 2 - Capture d'écran légendée d'un patch Pure Data mettant en œuvre un traitement sonore simple fondée sur un principe de modulation en anneau, comportant deux paramètres de contrôle (*Fréquence* et *Gain*).

La figure ci-dessus donne un exemple de patch réalisé dans le logiciel Pure Data qui modélise un traitement sonore simple à base de modulation en anneau [Figure 2]. Dans ce patch, l'objet *adc~* permet de recevoir le signal audio de la carte son sous la forme de blocs d'échantillons successifs. Ce signal est ensuite traité par les différents objets du graphe qui constituent le traitement spécifique de ce patch et permettent de fournir un résultat sous la forme d'un signal audio en sortie, renvoyé à la carte son via l'objet *dac~*. L'aspect de conception du patch met aussi en avant les paramètres qui seront exposés à l'utilisateur pour le contrôler durant son exécution. Dans ce cas, on pourra agir sur deux variables : la fréquence de modulation du signal audio d'entrée, et le gain du signal en sortie. Ces paramètres sont tous les deux manipulables en temps réel lors de l'exécution du patch par l'intermédiaire des objets de contrôle [Figure 2].

1.2. Collaboration

Si l'interaction se définit comme un échange entre plusieurs acteurs au sein d'un système, la collaboration peut se définir de manière générale comme une forme particulière d'interaction qui converge vers un but commun que partagent ses acteurs. Dans la littérature liée au travail coopératif assisté par ordinateur (TCAO)[16] on retrouve souvent le terme *groupware*. Ce terme est un néologisme inventé par Peter et Trudy Johnson-Lenz en 1978 qui désigne et comprend pour eux à la fois l'ensemble des processus et procédures d'un groupe de travail devant atteindre un objectif particulier, mais aussi les logiciels conçus pour faciliter ce travail de groupe [Peter & Trudy Johnson-Lenz, 1990]. Cette définition comporte donc des composants à la fois technologiques et humains. C'est un ensemble de méthodes et de techniques de travail en équipe, instrumentées par des outils logiciels conçus spécifiquement pour les soutenir. Ce type de logiciels, qui permettent à des utilisateurs reliés par un réseau de travailler en collaboration sur un même projet, sont désignés en français sous le nom de collecticiels, mais on utilise aussi les termes de logiciels de groupe ou encore logiciel multi-utilisateur par opposition aux logiciels traditionnels destinés à n'être manipulés que par un seul utilisateur à la fois (ou mono-utilisateur).

Plusieurs outils taxonomiques et conceptuels ont été conçus pour analyser les systèmes supportant le travail collaboratif. Ces systèmes peuvent différer selon leur nature, le domaine d'application auxquels ils répondent, les types d'interaction qu'ils soutiennent ou encore le type de fonctionnalités qu'ils offrent aux utilisateurs.

[16] Que l'on retrouve sous l'acronyme TCAO en français, ou encore CSCW en anglais pour *Computer-Supported Cooperative Work*. Notons que la notion de coopération diffère légèrement de la notion de collaboration dans le sens où elle comprend des processus de travail qui peuvent porter sur des tâches différentes.

1.2.1. Types d'interaction

C.A. Ellis, S.J. Gibbs et G. Rein proposent un premier outil de classification des interactions multi-utilisateurs fondé sur des critères de temps et d'espace [Ellis & al., 1991].

	Même moment	Moments différents
Même lieu	interactions directes en face-à-face	interactions asynchrones
Lieux différents	Interactions synchrones et distribuées	Interactions asynchrones et distribuées

Tableau 1 – Matrice d'Ellis en deux dimensions classant les types d'interaction en fonction d'un critère de temps et d'espace, traduit d'après [Ellis & al., 1991].

Cet outil, repris dans le [Tableau 1], se comprend comme une matrice en deux dimensions qui distribue, sur un axe à la fois temporel et spatial, les différents modes d'interaction entre les utilisateurs : synchrone ou asynchrone, colocalisée ou à distance :

- Dans la cellule « même lieu » et « même moment », on trouve les interactions directes, en face-à-face, on peut prendre l'exemple d'une salle de cours traditionnelle où des étudiants sont réunis en présentiel et peuvent interagir avec un professeur qui explique un traitement sonore en le projetant simplement sur un écran. Un autre exemple pourrait être celui d'une surface multi-tactile sur laquelle plusieurs personnes pourraient interagir pour faire de la musique à plusieurs sur la même interface.

- Lorsque les interactions se font dans un « même lieu » à des « moments différents », celles-ci sont exercées de manière asynchrone. On imagine alors dans ce cas des situations où un même patch, qui se trouve dans un studio par exemple, est édité à tour

de rôle par différents réalisateurs en informatique musicale à différents moments de la semaine ou du mois.

- Les interactions se situant dans des « lieux différents » à des « moments différents » s'effectuent de manière asynchrone mais sont aussi distribuées géographiquement. C'est le cas d'une interaction qui s'effectue en ligne au sein d'un forum de discussion par exemple. Plusieurs utilisateurs peuvent collaborer à distance pour résoudre un problème à plusieurs de manière itérative, donc non simultanée, et sans avoir besoin d'être physiquement l'un à côté de l'autre[17].

- Le dernier type d'interaction décrit par cet outil de classification est le cas où les personnes se situent dans des « lieux différents » aux « mêmes moments ». L'interaction a alors lieu en temps réel, même si les personnes se trouvent géographiquement éloignées les unes des autres, dans deux villes ou pays différents par exemple. C'est précisément ce type d'interaction que nous voulions étudier lorsque nous avons débuté notre travail de thèse.

Il nous faut néanmoins préciser que les notions de « moment » et de « lieu » présentées au sein de cette matrice sont souvent dans la pratique moins statiques ou binaires qu'elles ne le laissent paraître ici. Elles sont à notre avis plus à prendre sur des axes ou des échelles, qui peuvent alors être gradués grâce à des unités de temps (millisecondes, secondes, heures, jours…) et d'espace (mètres, kilomètres…). Nous pouvons considérer que des personnes se trouvent dans le « même lieu » si elles sont toutes réunies dans une même salle. Néanmoins, le type d'interaction pourra

[17] Ce cas sera abordé en détail dans l'étude des pratiques et des solutions collaboratives existantes que nous dresserons un peu plus loin dans ce document.

différer suivant si elles se trouvent à une vingtaine de mètres les unes des autres ou côte à côte. De la même manière, la notion de temps au niveau de l'interaction est, elle aussi, tout à fait subjective et relative au contexte spécifique de l'activité étudiée ou réalisée. S'il nous est permis de considérer dans certains contextes que deux actions ont lieu au « même moment » si elles interviennent à quelques secondes ou minutes d'intervalle ; dans des situations musicales de *live* par exemple, il convient souvent de considérer la notion de « moment » au niveau de la milliseconde pour évaluer la réelle simultanéité des événements.

1.2.2. Espaces de production, de coordination et de communication

La matrice d'Ellis nous fournit un premier outil qui nous permettra de classer les différentes solutions logicielles collaboratives en fonction du mode d'interaction qu'elles soutiennent. Pour nous aider à la formalisation des différents enjeux liés à la conception d'un système multi-utilisateur pour le patching audio, nous souhaitons introduire un autre outil conceptuel. Cet outil est le trèfle des systèmes multi-utilisateurs, qui permet d'analyser le comportement et de classer les fonctions principales que l'on trouve de manière générale au sein des solutions logicielles collaboratives.

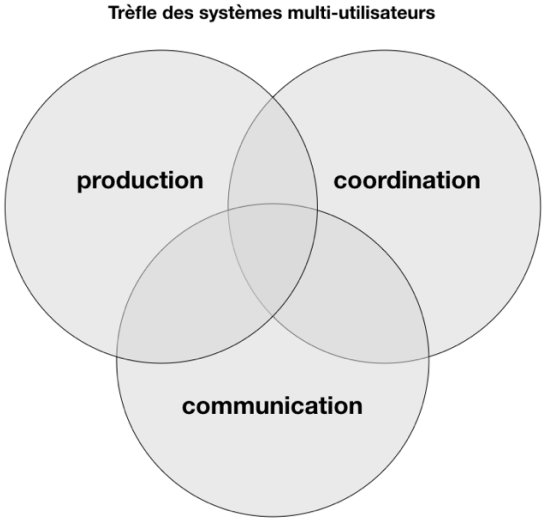

Figure 3 – Trèfle classant les différentes fonctions des systèmes multi-utilisateurs, repris et traduit depuis [Salber, 1995, p. 19].

D. Salber propose, en se basant sur le modèle conceptuel élaboré initialement par C.A Ellis et J. Wainer [Ellis & Wainer, 1994], [Salber, 1995], de classer les fonctions principales d'un système multi-utilisateur selon trois groupes ou espaces différents : l'espace de *production*, l'espace de *communication* et l'espace de *coordination* [Figure 3]. Ce modèle a été repris par le groupe de travail sur les systèmes coopératifs et porte aussi le nom de *modèle du trèfle de GT-SCOOP* [Beaudouin-Lafon & Coutaz, 1995].

Les trois espaces peuvent être compris comme suit :

- L'espace de ***production*** reprend le *modèle ontologique* élaboré par C.A Ellis et J. Wainer. Il désigne les objets accessibles, produits ou manipulés par le groupe et les opérations possibles sur ces objets [Ellis & Wainer, 1994] ; ce qui résulte de l'activité de groupe au sein d'un logiciel multi-utilisateur, qui « dénote l'œuvre tangible commune » [Salber, 1995, p. 19], par exemple un article, une partition, un morceau de musique, ou dans notre cas un traitement sonore. L'espace de production est aussi lié à la problématique du contrôle d'accès, c'est-à-dire à la définition des éléments de l'espace de production personnel qui restent locaux et ceux qui peuvent être accessibles au sein de l'espace de production d'un autre utilisateur connecté au même collecticiel.

- L'espace de ***coordination***, qui reprend le *modèle de coordination* d'Ellis et Wainer, est relatif à l'organisation entre les participants, aux relations entre leurs activités et à la gestion du flux de travail de manière générale. Il correspond aussi à la mise en œuvre de solutions à la fois logicielles et ergonomiques qui permettent aux utilisateurs de coordonner leur travail au sein de l'espace de production et/ou de communication. Parmi les enjeux spécifiques liés à cet espace on trouvera par exemple le fait de devoir fournir aux utilisateurs les moyens d'accéder aux différentes ressources ou documents proposés

au sein du logiciel (gestion des sessions), de gérer les cas de conflits générés par des accès concurrents aux données de l'espace partagé (*contrôle de la concurrence*), mais aussi de fournir des éléments de compréhension des actions de groupe, notamment via l'interface graphique du logiciel, pour aider chaque participant à situer ses propres actions par rapport à celles du groupe (mécanismes de *conscience de groupe*)[18].

- L'espace de ***communication*** permet aux utilisateurs du système d'échanger des informations. D. Salber note néanmoins que « le contenu sémantique de cette information concerne les acteurs communicants. Il est étranger au système qui se contente de servir de messager »[19]. Le cas typique d'une fonction comme celle-ci se retrouve par exemple dans une solution de messagerie instantanée, ou encore, dans le cas des pratiques de patching audio, dans les solutions qui permettent le contrôle des paramètres d'un patch à plusieurs, c'est-à-dire la communication des valeurs de jeu entre les participants.

Certains outils collaboratifs se fondent dans les trois espaces. C'est par exemple le cas de la suite d'outils de bureautique en ligne *Google Docs*, qui propose :

- Des *fonctions de production* qui permettent entre autres l'édition d'un texte.
- Des *fonctions de coordination*, que l'on trouve notamment dans le fait de pouvoir inviter quelqu'un sur un document, gérer de manière automatique la concurrence et les conflits d'édition du texte, ou encore offrir des solutions graphiques qui permettent aux utilisateurs d'obtenir des informations sur l'activité des autres utilisateurs sur le

[18] Ces différents aspects fonctionnels seront abordés plus en détail dans le chapitre dédié à la formalisation des enjeux liés au sujet.

[19] [Salber, 1995], p. 19.

document en temps réel, telles que des curseurs ou des sélections de couleurs différentes.

- Des *fonctions de communications* telles qu'une messagerie instantanée qui permet aux utilisateurs de communiquer via un envoi de messages en temps réel ou encore d'apposer des commentaires au texte et d'établir ainsi une communication asynchrone.

D'autres outils collaboratifs peuvent ne fournir qu'un seul espace, par exemple l'espace de production ou l'espace de communication ou encore apporter simplement des fonctions de coordination. D. Salber souligne aussi que ces fonctions ou ces espaces ne sont pas figés et qu'ils peuvent varier au cours de l'interaction et de l'exécution du système [Salber, 1995, p. 19]. Ils peuvent aussi varier et se composer de différents éléments suivant le type d'activité sur lequel se porte la collaboration.

1.3. Patching collaboratif

Un patch peut être vu à la fois comme un document statique (qui peut être sauvé en tant que fichier sur le disque), mais aussi comme un document dynamique dans le sens où, chargé au sein de l'environnement dans lequel il a été créé, il peut être exécuté pour créer et contrôler des processus musicaux en temps réel. Si on essaye de transposer les fonctionnalités offertes par un environnement mono-utilisateur de patching à une solution logicielle du même type, mais qui permet d'effectuer des opérations similaires à plusieurs, quels sont les objets et les données que l'on cherche à partager ? Quels types d'opérations pourraient être permises par ce système et quels types d'activités mises en commun est-on à même d'envisager qu'il puisse supporter ? Nous avons discerné deux types d'activités principales au sein de ces environnements, l'activité de conception du patch et l'activité de jeu. Dans le cadre d'une édition collaborative, les données à partager correspondent aux informations relatives au modèle du patch c'est-à-dire à

sa structure. Les utilisateurs s'intéressent dans ce cas à la co-construction d'un même instrument ou plus généralement à la manière dont est mis en œuvre un traitement au sein d'un patch, on se place donc ici à un niveau algorithmique. Dans le cas d'un jeu partagé, les utilisateurs cherchent seulement à utiliser le même instrument préexistant à plusieurs, on voudra alors transmettre en temps réel l'état du programme aux autres participants et non-plus sa structure. Il existe aussi des cas intermédiaires, comme dans les pratiques de *live coding*, où les utilisateurs se servent de l'instrument dans le cadre d'une performance tout en le créant, en l'éditant en même temps qu'ils en jouent. Dans ce cas c'est à la fois la structure du document et son état interne que l'on cherchera à partager.

Afin d'avoir une vision plus concrète des différentes problématiques liées au travail collaboratif à partir des logiciels de patching existants, il nous faut les observer de plus près en étudiant les pratiques déjà à l'œuvre et les solutions logicielles qui permettent de les soutenir.

2. Études des pratiques et solutions logicielles existantes

Dans un premier temps nous nous concentrerons sur les pratiques et outils logiciels permettant une édition collaborative asynchrone de patch, nous y aborderons de manière concrète les problèmes posés par l'approche et notamment ceux liés à l'édition concurrente de document. Nous nous intéresserons ensuite aux systèmes permettant une collaboration synchrone dans le cadre des pratiques de patching, à la fois au niveau d'un partage de l'édition et du partage de l'exécution du patch, nous nous attarderons alors sur la manière dont sont conçus de tels systèmes d'un point de vue architectural et sur les types de données qui y sont partagées.

2.1. Édition collaborative asynchrone

Nous employons tous des techniques et des solutions logicielles pour collaborer avec d'autres personnes sur des projets musicaux (ou autres) dans notre pratique quotidienne. La condition préalable à toute collaboration est le partage des données. Dans le cas de l'édition collaborative asynchrone de patch, l'élément partagé entre les utilisateurs est le patch, en tant que document statique. Ce partage peut s'effectuer simplement par copie, par courrier électronique ou encore via un transfert par clef USB. Les utilisateurs disposant des mêmes données stockées au sein d'un fichier peuvent alors apporter des modifications chacun de leur côté au document. Le problème bien connu lié à ce procédé est que les modifications apportées parallèlement par les utilisateurs ne sont pas synchronisées entre-elles et on assiste alors à une démultiplication des versions du même document source, et à une perte de la notion de document référent auquel se fier. Dès lors, quelles solutions logicielles peuvent venir soutenir ces pratiques ? Une première solution consiste à centraliser l'information en ligne, c'est-à-dire le patch, de façon à pouvoir déterminer une version référente du document. Le premier cas de collaboration asynchrone que nous avons choisi d'étudier entre dans ce cadre. C'est une pratique au sein de laquelle des personnes usent d'un forum de discussion comme support logiciel permettant le partage d'un document commun pouvant être édité par plusieurs personnes. Une autre solution consiste à maintenir des copies indépendantes du document chez chacun des participants puis à les synchroniser entre-elles grâce à des outils permettant la gestion de version. Nous étudierons cette solution dans un second temps.

2.1.1. Cas d'étude du forum en ligne

Les forums de discussion hébergés en ligne constituent des espaces numériques où des personnes peuvent se retrouver virtuellement afin d'échanger des idées, poser des questions pour résoudre des problèmes spécifiques relatifs à leurs propres pratiques, ou encore

simplement discuter sur des thèmes qui les rassemblent. Il existe des forums ou des outils similaires dédiés à de nombreux domaines d'application. Dans le cadre des pratiques temps réel, on peut citer la *mailing-list* de Pure Data[20] ou encore le forum dédié au logiciel Max, divisé en plusieurs sous-groupes et hébergé sur le site de l'éditeur Cycling'74[21]. Au sein d'une entrevue réalisée auprès de plusieurs utilisateurs actifs du logiciel Max, Andrew Benson souligne à quel point il représente un outil fédérateur et constitutif de la communauté d'utilisateurs du logiciel Max ainsi qu'un lieu privilégié pour la résolution de problèmes et l'apprentissage collectif [Benson, 2012]. Nous vérifions d'ailleurs cela aussi du point de vue de notre propre pratique. Le forum a constitué pour nous une ressource essentielle à l'apprentissage du langage Max, et nous a aussi aidé en tant que développeur à de nombreuses reprises, quand il s'agissait par exemple d'obtenir des clarifications sur les spécificités de son API[22], notamment dans le cadre du développement de la bibliothèque *HOA*[23] [Colafrancesco & al., 2013], [Sèdes & al., 2014]. Les usages et les avantages des forums en ligne, notamment en matière d'apprentissages individuels et d'apprentissages collaboratifs, et en particulier dans le milieu éducatif, sont aujourd'hui bien documentés [Henri & Lundgren-Cayrol, 2001], [Walckiers & De Praetere, 2004], [Jelmam, 2010]. Ce que nous avons cherché à savoir dans le cadre de la réalisation de cette étude, c'est s'il existait de nouveaux usages au sein des forums, non-pas seulement en matière d'apprentissages collectifs mais surtout quant au partage du faire.

2.1.1.1. Hippie patching

En plus de pouvoir contenir du texte, les forums offrent aussi aux utilisateurs la possibilité d'associer des ressources telles que des images ou des documents à leurs messages. Certaines

[20] https://lists.puredata.info/listinfo/pd-list

[21] https://cycling74.com/forums/

[22] Application Programming Interface.

[23] http://hoalibrary.mshparisnord.fr/.

personnes ont donc logiquement eu l'idée de s'emparer de ce medium en le détournant légèrement afin d'en faire un réel espace collaboratif de travail. L'idée générale est assez simple, elle consiste à se servir du forum comme d'un espace de stockage centralisé, pour construire un patch à plusieurs mains de façon itérative, par messages interposés.

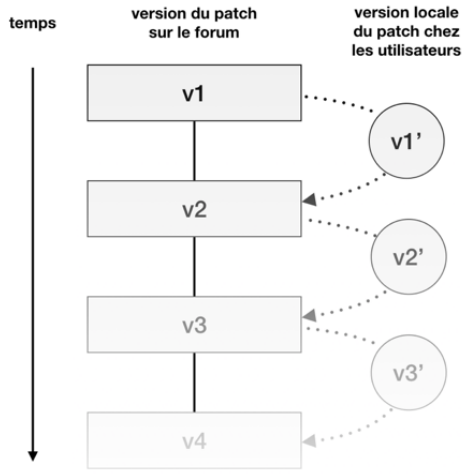

Figure 4 – Schéma présentant le concept de *chaîne de patch* mis en place par des utilisateurs au sein du forum de Max. Le schéma se lit de haut en bas, à chaque itération le patch est téléchargé par un utilisateur sur sa machine, modifié puis posté dans un nouveau message. On considère que la version du patch est incrémentée à chaque itération.

Un premier utilisateur poste un message contenant un patch, puis un autre lui répond par un second message accompagné d'une nouvelle version de ce même patch comportant les modifications qu'il a souhaité y apporter. L'opération peut alors se répéter indéfiniment, créant ainsi une chaîne constituée du même patch, édité de façon séquentielle par plusieurs membres du forum [Figure 4].

Les premières traces relatives à ce genre de pratiques que nous avons pu relever remontent à 2010, lorsqu'un utilisateur, surnommé *DIGIOLOGY*, crée un fil de discussion intitulé *collaborative-patching* qui commence comme suit :

> Digiology : *Je me demande si quelqu'un a déjà essayé de faire du patching collaboratif en ligne, peut-être à travers ce forum. Nous pourrions commencer avec quelqu'un qui publie un patch, puis il serait modifié de manière itérative par les membres du forum. Qu'en pensez-vous ?*
>
> ---
>
> Brendan McCloskey*: C'est une excellente idée, et une utilisation novatrice du forum, [...] je suis partant.* [24]

Il a conduit dès le lendemain à la création de trois fils de discussion isolés[25], dédiés à la mise en place et à la concrétisation de cette démarche qui ont eux-mêmes mené à l'élaboration collaborative d'un patch par cinq utilisateurs différents au sein du forum. Une capture de l'état final de ce patch est exposée un peu plus tard dans ce texte. Cette première initiative n'est pas isolée, un autre fil de discussion, créé près d'un an après et intitulé "*Hippie patching*"[26], a par exemple suscité l'engouement de plus de 21 utilisateurs différents et généré 75 réponses entre le 28 juillet et le 23 août 2011[27] [XH9O, 2011].

---

[24] Tiré de [Digiology, 2010a], traduit de l'anglais : « Digiology: I'm wondering if anyone has tried collaborative patching online perhaps through this forum. We could conceivably start with someone posting a patch and then it being modified iteratively by members of the forum. What do you guys think?

Brendan McCloskey: This is a great idea, and a novel use of the forum, […] count me in. »

[25] Les deux premiers sont destinés à la création de patch audio, un pour des utilisateurs de niveau intermédiaire (https://cycling74.com/forums/mspaudio-collaborative-patch-intermediate), un autre pour des utilisateurs avancés (https://cycling74.com/forums/mspaudio-collaborative-patch-advanced), le troisième est dédié à la création d'un patch manipulant de la vidéo (https://cycling74.com/forums/jittervideo-collaborative-patch). Il est à noter que seul le premier fil de discussion a été alimenté par la suite.

[26] Le terme "*hippie patching*" est sans doute un clin d'œil à l'expression "Happy patching" employée par Andrew Pask pour conclure chaque vidéo de la série de tutoriel pour le logiciel Max intitulée "*Did You Know*" (https://www.youtube.com/watch?v=sEbyGLkVnrk&index=1&list=PL71A2325EEEFC1923). Le choix d'employer ce terme comme titre à ce fil de discussion semble pour nous appuyer le fait que les auteurs se placent consciemment au sein d'une culture commune et dans une démarche d'apprentissage par le faire ensemble.

[27] Certains utilisateurs insatiables sont même allés jusqu'à créer un nouveau fil de discussion peu de temps après celui-ci intitulé *Revenge of Hippie Patching* : https://cycling74.com/forums/revenge-of-hippie-patching.

> XH9O : *Peut-être que je suis malade, mais juste avant de m'endormir j'ai eu l'idée de créer une chaîne de patch. J'entends par là que nous patchions ensemble ; sans aucun but précis, juste pour s'amuser* [28].

Ces différents fils de discussions sont pour nous des ressources précieuses et essentielles à étudier, d'abord pour leur dimension historique, mais surtout car ils font émerger, de manière expérimentale, les attentes, les problématiques et les singularités du travail à plusieurs et de la création d'une œuvre commune dans le cadre spécifique de la pratique du patching audio. Ils définissent le protocole à mettre en place, les précautions d'usage à prendre, et les spécifications d'un tel système du point de vue de l'utilisateur du logiciel. C'est pourquoi il nous a semblé particulièrement pertinent d'en proposer une lecture croisée et critique dans ce cadre.

Contrairement à des travaux précédents menés sur les forums en ligne [Jelman, 2010], nous ne nous situons pas uniquement dans une étude portant sur la co-construction de la connaissance mais bien dans une étude de la co-construction d'un même objet ou instrument que constitue le patch, d'autre part il est aussi important de préciser que cette brève étude porte sur des initiatives spontanées des utilisateurs, étudiées *a posteriori* et de l'extérieur.

Juste après avoir lancé son idée, l'auteur du second fil de discussion définit certaines règles et contraintes que chaque participant se devra alors de respecter [XH9O, 2011] :

> XH9O : *[…] règles :*
> *1) Le patch est posté et stocké dans ce fil de discussion.*

---

[28] *Tiré de [*XH9O, 2011], traduit de l'anglais : « *XH9O*: Maybe I´m just mad, but right before falling to sleep I had the idea of a user chain patch. means, we patch together. without any great concept, just for fun ».

> *2) si vous ajoutez quelque chose, postez la dernière version et faites un petit commentaire.*
> *3) SEULEMENT 5 nouveaux objets (objets mathématiques et* trigger *non-compris) et 15 messages par tours.*
> *4) vous pouvez avoir à nouveau la main quand au moins 2 autres utilisateurs ont ajouté quelque chose après votre dernier tour.*
> *5) vous pouvez apporter des modifications dans les mêmes conditions (5 objets / 15msg - mais il seront retirés de votre [crédits d'ajout], cela signifie que vous ne pouvez ajouter que 2 nouveaux objets si vous en modifiez 3 etc.)*[29]

Les points trois et cinq sont pour nous purement arbitraires et visent, semble-t-il, simplement à fixer et limiter la taille de chaque contribution afin de permettre la coordination des actions et faciliter la construction progressive du patch[30]. Le premier point met quant à lui l'accent sur la nécessité d'avoir accès à la même ressource ; le second porte sur la documentation de la pratique ; le quatrième point traite enfin du problème du contrôle de la concurrence. Nous revenons sur ces trois derniers points dans les parties suivantes.

2.1.1.2. Gestion d'une ressource commune

Un accès centralisé à la même ressource est primordial dans ce type d'initiative, en filigrane cela signifie aussi le fait que chacun doit pouvoir voir et/ou entendre la même chose, ou encore que le patch ait le même comportement chez tout le monde. Ce cas ne semble pas poser de

---

[29] Tiré de [XH9O, 2011], traduit de l'anglais « XH9O: *[…] rules:*
1) the patch is posted and carried in this thread
2) if you add something, post the newest version and make a short comment
3) ONLY 5 new objects (asides triggers and math objects) and 15 messages per run
4) you can have a new run when at least 2 other users added something after your last turn
5) you can modify under the same conditions (5 objects/15msg - but it is stolen from your ADD NEW-credits, means like you can add only 2 new when modifying 3 objects etc.) ».

[30] Il est à noter que certaines de ces règles seront d'ailleurs vite discutées et enfreintes au cours de l'évolution de ce fil de discussion.

problème particulier dans la mesure où l'accès à la seule ressource partagée que constitue le patch se fait directement sur le serveur central qui héberge le forum. Un patch Max ou Pure Data peut néanmoins dépendre de beaucoup d'autres éléments pour fonctionner correctement, il était donc à prévoir que certains utilisateurs rencontrent certaines difficultés :

> CH : *@stefantiedje: [message d'erreur :] "newobj: mod: No such object" :)*
> *Je pense qu'il serait bon de s'en tenir aux objets standard*[31].

Cette dernière citation expose par exemple un message d'erreur qui apparaît au chargement du patch chez un utilisateur, et qui est imputable à l'emploi d'un objet externe non-standard dont il ne dispose pas. Le cas des objets externes n'est pas la seule source pouvant générer des incohérences dans ce que voient ou entendent les différents utilisateurs du patch. Il a donc été décidé de façon collégiale dans le cadre de cette expérience, que le patch devrait donc se suffire à lui-même et ne pas dépendre de ressources externes telles que des objets externes[32], des abstractions, ou encore des fichiers sonores. Certains utilisateurs sont même allés plus loin en définissant le type de configuration audio que chacun devrait adopter pour garantir que le rendu sonore soit bien le même partout [XH9O, 2011].

[31] Tiré de [XH9O, 2011], traduit de l'anglais : « CH: *@stefantiedje: newobj: mod: No such object :)* I think it's better to stick to standard objects ».

[32] Les logiciels Max et Pure Data disposent d'une API qui permet d'étendre les fonctionnalités standards du logiciel en programmant de nouveaux modules (objets externes) qui peuvent être dynamiquement chargés au sein de l'environnement de base.

2.1.1.3. Documentation des actions

Le fait de documenter les modifications apportées par chacun dans ce type de pratique est primordial afin que les utilisateurs puissent prendre conscience des changements effectués par les autres et améliorer l'interaction [McCaffrey, 1998].

> Digiology : *Vous devez expliquer la nature de votre modification ou son effet sur le fonctionnement du patch et comment l'utilisateur peut interagir avec lui (si nécessaire)*[33].

Les commentaires des utilisateurs portant sur le patch sont souvent d'une importance capitale car ils témoignent de leurs intentions et permettent de répondre à des questions comme :

- Qui a ajouté tel objet ?
- Pourquoi tel utilisateur a ajouté telle fonctionnalité au patch de cette manière ?
- Quelle était son intention au moment de rajouter telle fonctionnalité au patch ?
- Quels étaient les prochaines étapes de développement qu'il envisageait au moment de faire tel ou tel choix ?

Les différentes réponses des utilisateurs constituent ainsi des métadonnées offrant de la pérennité au document co-créé et une meilleure compréhension du processus de construction par les utilisateurs qui peuvent s'y référer dans la durée. Si certains de ces commentaires ont été insérés directement au sein du patch, beaucoup se retrouvent seulement sur le forum et sont donc malheureusement détachés du document, les rendant inaccessibles depuis l'interface du logiciel Max, qui ne permet donc pas à elle seule de rendre compte de l'activité qui s'y est déroulée.

---

[33] Tiré de [Digiology, 2010b], traduit de l'anglais : « Digiology: You should explain what your modification is or what its effect is on how the patch works and how the user may interact with it [if they need to at all] »

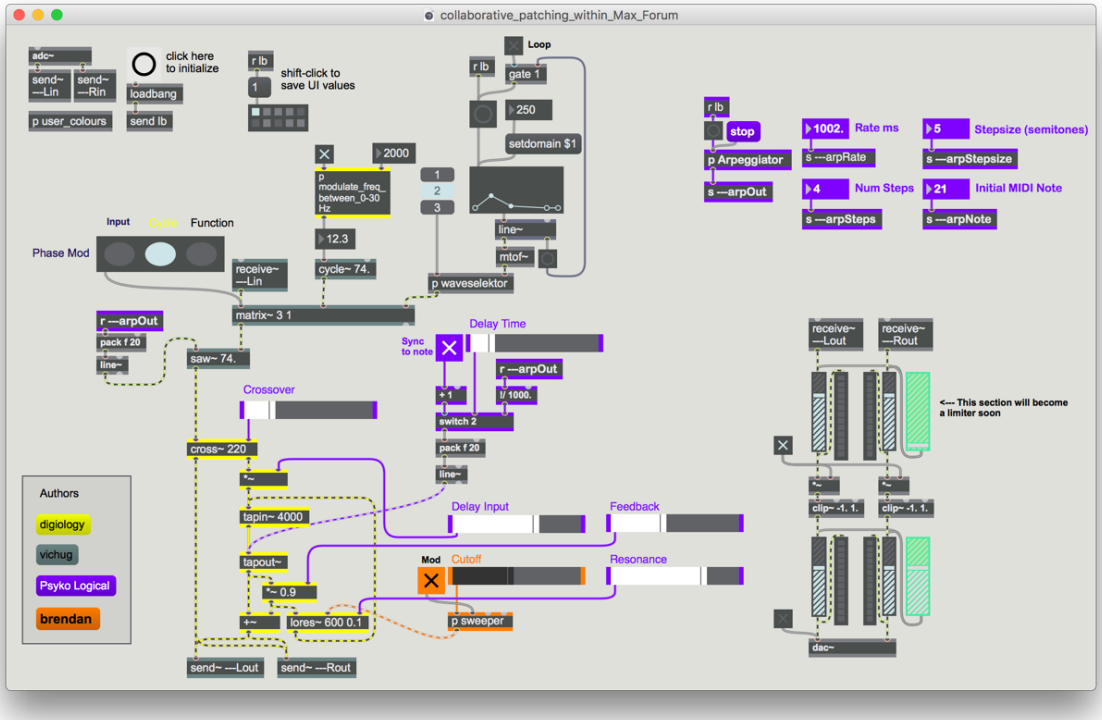

Figure 5 – Capture d'écran de la dernière itération du patch Max élaboré de manière collaborative par plusieurs utilisateurs du forum de Max, consécutivement à la création du fil de discussion intitulé *collaborative-patching* [Digiology, 2010b]. Les contributions respectives de chacun des auteurs ont été marquées par eux-mêmes en colorant d'une couleur spécifique le contour des objets qu'ils ont créés ou modifiés.

Un autre type de documentation, non-textuelle cette fois-ci, peut aussi être ajouté grâce notamment aux couleurs des objets. C'est ce qu'ont par exemple fait instinctivement les utilisateurs ayant contribué au fil de discussion [Digiology, 2010b] présenté à la [Figure 5], dans lequel on peut discerner l'apport des différents auteurs grâce à l'attribution de couleurs spécifiques symbolisant leurs contributions respectives.

2.1.1.4. Gestion de la concurrence

Le fait de pouvoir contrôler les accès concurrentiels au document est une dimension essentielle de l'édition collaborative et se révèle dans ce cas de collaboration asynchrone aussi un problème obligatoire à considérer. En effet, la technologie présente au sein du forum ne permet pas

d’éditer directement un patch en ligne. Pour ce faire, chaque personne désireuse d’apporter sa contribution au document commun doit donc télécharger celui-ci depuis la dernière itération sur le forum sur sa propre machine puis l’éditer en local sur son application de bureau Max avant de la téléverser à nouveau sur le forum en créant un nouveau message, comme montré à la [Figure 4]. Le document est donc destiné à n’être édité que par une seule personne à la fois pour que la chaîne reste cohérente.

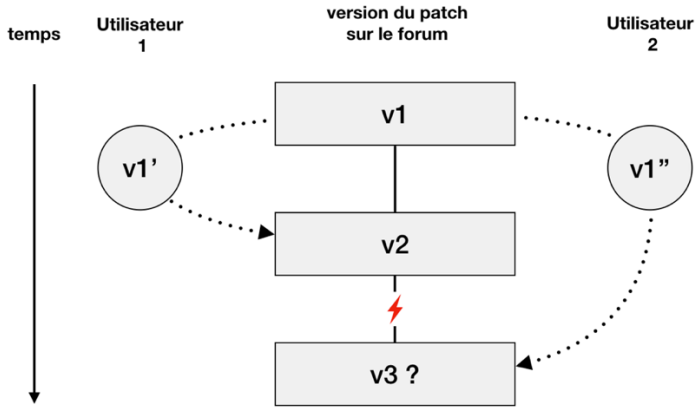

Figure 6 - Schéma représentant la structure d’une chaîne de patch dans laquelle deux utilisateurs éditent simultanément la même version de départ d’un patch puis les postent chacun leur tour sur le forum, générant une rupture de la linéarité de la chaîne.

Néanmoins, aucune solution logicielle ne vient prévenir cette édition simultanée, et plusieurs personnes peuvent donc accéder au même moment au patch en le téléchargeant, puis le modifier chacun de leur côté simultanément comme montré à la [Figure 6]. L’auteur initial du premier fil de discussion note alors son inquiétude :

> Digiology : *Je m'inquiète juste que quelqu'un puisse prendre une itération, la regarder pendant 10 minutes pour comprendre ce qui se passe, puis passer 15 minutes à la modifier. Je ne veux pas que leur travail soit gâché ou pire que quelqu'un poste une itération qui ignore*

> *complètement la précédente parce qu'elle a été postée pendant qu'ils codaient la leur. Ce serait très frustrant de perdre son travail comme ça*[34].

Dans le schéma présenté plus haut, deux utilisateurs ont téléchargé depuis le même message du forum, le patch à sa version *v1* et l'ont tous deux édité. Quand le premier utilisateur poste son message avec le nouveau patch qu'il vient de modifier, on considère que la version du document doit être incrémentée et celui-ci passe donc logiquement à la version *v2*. Le second utilisateur poste ensuite sa propre contribution sous la forme d'un nouveau message contenant ses modifications apportées à la version initiale du document. On se retrouve alors dans une situation qui génère une rupture de la linéarité de la chaîne de document dans la mesure où, du point de vue du deuxième utilisateur la version qu'il téléverse correspond à la *v1+1* soit la *v2*, mais du point de vue du forum la dernière version devrait être la *v3*. Cette situation mène alors à des problèmes pratiques. En effet, un utilisateur qui rejoindrait le fil de discussion à ce moment-là pourrait très bien ne pas s'apercevoir de la modification concurrente et partir de la dernière version téléversée sur le forum pour y apporter sa contribution ; les modifications apportées par la *v2* sur le forum seraient alors ignorées et perdues. Ces problèmes ont bien été identifiés par les membres de ces fils de discussion qui ont d'ailleurs anticipé certaines solutions possibles permettant d'y faire face :

> Digiology : *Dans le cas peu probable où il y a un conflit et deux itérations sont postées en même temps, la prochaine personne peut choisir de fusionner celles-ci ou d'en choisir une. Plus généralement, les gens peuvent simplement choisir d'annuler la modification de*

[34] Tiré de [Digiology, 2010a], traduit de l'anglais : « *Digiology:* I'm just concerned that someone might take an iteration, look at it for 10 minutes to figure out what's going on, and then spend 15 minutes modifying it. I don't want their work to be wasted or worse have someone post an iteration that completely ignores the previous one because it was posted while they were coding theirs. It would be very frustrating to have your work lost like that »

> *l'itération précédente ou de commencer à partir d'une itération beaucoup plus ancienne s'ils le veulent. Néanmoins, il serait bien d'éviter de le faire autant que possible.*[35].

Dans le cas où l'utilisateur prend la mesure de la situation il peut alors choisir d'ignorer volontairement la version précédente ou alors de réincorporer les modifications de la version précédente avec celle de la version plus récente manuellement pour générer une nouvelle version comportant l'ensemble des modifications, opération qui peut s'avérer délicate à réaliser dans certains cas. Ce procédé est plus connu sous le nom de fusion (ou *merge* en anglais) dans le domaine de la gestion de version informatique[36].

Comme dans la plupart des forums en ligne, celui de Max permet aussi aux utilisateurs d'éditer leur propre réponse après qu'elle ait été postée. Certains contributeurs ont donc modifié le patch contenu dans leur réponse après l'avoir téléversé, ce qui peut encore générer des incohérences dans l'historique du forum et d'éventuels conflits. Un utilisateur, conscient du problème que cela pourrait poser, prévient les autres participants au fil de discussion :

> R Σ N Σ GΔD Σ RΔJΔ: *J'ai changé mon patch et l'ai posté à nouveau, maintenant je n'y touche plus (avec un peu de chance, personne n'y a encore touché, mais si vous aviez déjà jeté un œil avant de voir ce nouveau message, recopiez et utilisez la nouvelle version disponible dans la version éditée du précédent message).*[37]

---

[35] Tiré de [Digiology, 2010b], traduit de l'anglais : « *Digiology:* In the unlikely event that there is a conflict and two iterations are posted together the next person can choose to merge these or choose one. More generally people can simply choose to undo the previous iteration's modification or start from a much earlier iteration if desired. It would be nice to avoid doing this as much as possible though ».

[36] La prochaine section, dédiée aux logiciels de contrôle de version, s'attardera plus en détail sur ces notions.

[37] Tiré de [XH9O, 2011], traduit de l'anglais : « R Σ N Σ GΔD Σ RΔJΔ: I changed my patch and then repasted, now I'll leave it alone (hopefully no one modified yet, but if you did take a look before seeing this new post, recopy and use the new version now available in the edited previous post) ».

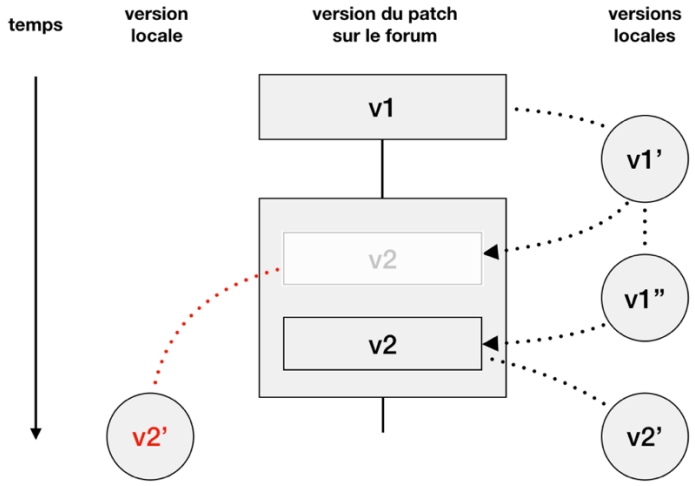

Figure 7 – Schéma représentant la structure d'une chaîne de patch dans laquelle un utilisateur édite une première fois un document, le téléverse dans un nouveau message sur le forum, puis le modifie à nouveau avant de le téléverser dans son précédent message en éditant simplement ce dernier.

Ces situations arrivent en fait plus souvent qu'il n'y paraît dans ce type de pratique. Si un utilisateur s'aperçoit par exemple qu'il a fait une petite erreur dans le patch qu'il vient de mettre en ligne, il peut être tenté de la corriger rapidement puis de reposter le patch comme si l'erreur n'avait jamais existé [Figure 7]. Dans le domaine du développement logiciel cette pratique est souvent appelée un fixe rapide (ou *quick-fix* en anglais). Dans ce cadre, elle permet de ne pas créer de bruit inutile au sein du fil de discussion en annonçant une nouvelle version qui vise seulement à résoudre des problèmes mineurs et n'apportent pas d'ajout de fonctionnalités majeures. En revanche, elle pose encore une fois le problème de l'intégrité de la version du document et souligne l'importance que dans un espace d'édition partagé, tout le monde puisse se référer à la même version d'un document.

Plusieurs solutions ont été envisagées, sous la forme de règles à respecter par les membres, pour limiter ces frictions et gérer les cas de conflits liés à l'édition concurrente, comme le fait

d'attendre un certain nombre d'itérations avant de pouvoir à nouveau intervenir sur le patch afin laisser du temps et la place d'intervenir à d'autres personnes :

> XH9O: *vous avez le droit à un nouveau tour quand au moins 2 autres utilisateurs ont ajouté quelque chose après votre dernier tour*[38].

Ou encore des solutions encore plus restrictives telles que la réservation d'un créneau horaire au sein duquel un seul utilisateur aurait le droit d'intervenir :

> Digiology: *Nous allons avoir besoin d'un protocole afin d'éviter que les gens travaillent simultanément sur une seule itération. Peut-être quelque chose comme:*
> *- Après une qu'une itération soit postée, quelqu'un répond pour réserver un créneau horaire afin de travailler sur l'itération [même si ce n'est qu'une petite modification].*
> *- Après avoir « réservé », ils doivent poster leur itération avant, disons, une heure [ou] exprimer leur intention d'apporter une modification. Ne pas le faire permet à quelqu'un d'autre de réserver.*
> *- Après que l'itération est posté, n'importe qui est libre de réserver un nouveau créneau*[39].

Les notifications étant asynchrones et les messages parfois éloignés de plusieurs heures ou jours dans le temps sur les forums, il est donc impossible de savoir précisément qui a effectivement la main sur le document. Les solutions mises en place dans le cadre de ces différents fils de discussion s'apparentent à un système de verrouillage (*lock*), mais non robuste dans la mesure

---

[38] Tiré de [XH9O, 2011], traduit de l'anglais : « XH9O: you can have a new run when at least 2 other users added something after your last turn. »

[39] Tiré de [Digiology, 2010a], traduit de l'anglais : « Digiology: *We would need some protocol in order to avoid people working concurrently on one iteration. perhaps something like:*

- After an iteration is posted someone replies to book a time slot to work on the iteration [even if it's only a small modification].

- After 'booking' they must post their iteration within [lets] say an hour [of] expressing their intent to modify. Failure to do so allows someone else to book.

- After the iteration is posted someone is free to book a slot. »

où il n'est nullement soutenu ni garanti au niveau logiciel. Tout repose simplement sur le bon usage des utilisateurs et le respect de certaines règles fixées par eux-mêmes.

Ces solutions temporaires souffrent donc de problèmes inhérents. En effet que se passe-t-il par exemple si un utilisateur réserve un accès au patch et choisit de le garder plus longtemps que prévu ? Il bloque alors l'ensemble des utilisateurs qui pourraient avoir envie d'y contribuer au même moment, les forçant alors à attendre inutilement que ce dernier leur rende la main. Elles restent d'autre part perméables aux conflits.

2.1.1.5. L'apprentissage induit par la pratique

Certains utilisateurs de ce fil de discussion ont rapproché cette démarche de celle du « cadavre exquis » inventé par les surréalistes. Cette pratique est un « jeu qui consiste à faire composer une phrase, ou un dessin, par plusieurs personnes sans qu'aucune d'elles ne puisse tenir compte de la collaboration ou des collaborations précédentes » [40]. Si ces deux expériences diffèrent bien dans la pratique - l'intérêt principal étant justement ici de voir les différentes contributions des utilisateurs - nous pouvons néanmoins rappeler que dans ces deux initiatives, la notion de jeu et d'amusement y est prépondérante et que si l'apprentissage n'était pas le but recherché en premier lieu, il a bien été induit : « Bien que, par mesure de défense, parfois cette activité [de jeu] ait été dite par nous « expérimentale », nous y cherchions avant tout le divertissement. Ce que nous avons pu y découvrir d'enrichissant sous le rapport de la connaissance n'est venu qu'ensuite »[41]. Dans un commentaire laissé sur un fil de discussion créé un peu plus de deux ans après le premier cité dans ce cadre, un membre du forum résume cette initiative ouverte et ludique en soulignant l'intérêt particulier du point de vue de l'apprentissage que représente le

[40] *Dictionnaire abrégé du surréalisme*, Galerie des Beaux-Arts, In-8° broché, Paris, 1938.

[41] A. Breton, « L'un dans l'autre », *in Médium : communication surréaliste*, n° 2, février 1954, p. 17.

fait de pouvoir partager l'ensemble du processus de création et non-pas seulement un document dans son état fini :

> MARK DURHAM : *Il y a quelque temps, il y a eu un certain nombre de fils de discussion intitulés "Hippie Patching", une sorte de patch collaboratif avec un concept que l'on pourrait résumer par "ajoute quelque chose et fais passer". Je suis sûr que beaucoup d'entre vous s'en souviennent ou ont même participé. Que vous trouviez que ce soit un succès ou non n'est pas vraiment le sujet ici, mais certains utilisateurs ont commenté à l'époque à quel point il leur était utile de voir les choses se développer d'une manière aussi ouverte - quelque chose avec laquelle je suis d'accord. Si vous êtes capable de suivre le développement de quelque chose, vous n'apprendrez pas seulement comment faire de la rétro-ingénierie, mais vous aurez un aperçu du processus de pensée qui a eu lieu au cours de la création ... nettement plus utile.*[42]

Nous avons recensé de nombreux messages comme celui-ci, d'utilisateurs déclarant avoir beaucoup appris des autres grâce à cette démarche particulière :

> FRID : *la raison pour laquelle j'aime cette idée vient du fait qu'elle permet d'apprendre des autres d'une manière plus directe que sur le forum et à voir de nouvelles approches faites par d'autres* [43].

Il est aussi à noter qu'ils proviennent aussi d'utilisateurs débutants, parfois juste spectateur de l'expérience, qui affirment avoir beaucoup appris de la part d'utilisateurs plus expérimentés :

---

[42] Tiré de [Durham, 2012], traduit de l'anglais : « *MARK DURHAM:* A while ago there were a couple of threads here titled "Hippie Patching", a kind of collaborative patch with an add a bit and pass it on idea. I'm sure many of you remember these or even took part. Whether you think this was a success or not isn't really the point here, but some users commented at the time about how useful it was to see things developed in such an open way - something I for one agreed with. If you are able to follow the development of something you don't just learn how to reverse engineer, but you gain an insight into the thought process of creation... arguably more useful. »

[43] Tiré de [Digiology, 2010a], traduit de l'anglais : « the reason why I like the idea is because of the ability to learn from others in a more direct way than the forum and seeing new approaches made by others ».

> EDSONEDGE : *Même si je n'avais pas l'impression d'avoir beaucoup à ajouter, c'est assez instructif pour nous, débutants, de voir des programmeurs Max plus expérimentés développer des patchs de manière si ouverte :)* [44]

L'analyse de ces différents fils de discussion semblent donc confirmer aussi ce que disaient F. Henri et K. Lundgren-Cayrol à propos du forum, à savoir qu'il est « la technologie privilégiée de l'apprentissage collaboratif en ligne » [Henri & Lundgren-Cayrol, 2001].

2.1.1.6. Bilan de l'étude sur le forum

Les différents fils de discussion que nous venons de présenter et de commenter plus haut mettent à jour une pratique collaborative partant d'une initiative originale et permettant l'édition asynchrone d'un patch par plusieurs utilisateurs, distribués géographiquement, et regroupés autour d'un forum hébergé en ligne [Figure 4]. Ils expriment pour nous le besoin des utilisateurs en matière de collaboration dans le domaine du patching audio, l'engouement de ces derniers pour ce nouveau genre d'interaction et mettent en exergue les aspects ludiques et les enjeux pédagogiques liés à ce type de démarches.

Ils permettent aussi d'entrevoir de manière concrète et liminaire un certain nombre de problématiques propres à ce type de pratiques, à savoir :

- La nécessité d'adapter l'interface du patch pour qu'elle puisse faire ressortir des informations sur les différents contributeurs et la nature de leur contribution. Le fait de travailler à plusieurs sur un document change le rapport des utilisateurs à l'interface qui

[44] Tiré de [XH9O, 2011], traduit de l'anglais : « *EDSONEDGE :* Even though I didn't feel I had much to add, it is quite educational for us noobs to see more experienced max programmers developing patches in such an open way :) »

doit alors s'adapter pour refléter le travail collectif. Nous retenons dans ce cadre le fil de discussion [Digiology, 2010b], au sein duquel les utilisateurs ont pris l'initiative de donner une couleur de contour différente aux objets pour permettre de distinguer plus clairement les contributions de chacun.

- Le besoin de définir certaines règles pour orchestrer le travail à plusieurs et l'importance d'une autodiscipline de la part des utilisateurs.
- La constitution d'un historique des changements par la documentation progressive et itérative des modifications apportées au document. Dans ce cas, l'historique peut être consulté simplement par la lecture du fil de discussion sur le forum.
- De savoir dans quelle mesure chacun doit pouvoir voir ce que voient les autres au sein du patch et qu'il puisse être exécuté de la même manière chez tous les participants.
- Le fait de devoir définir des canaux spécifiques de communication pour permettre à différents groupes d'interagir de façon autonome, ce que S. Greenberg définit comme la gestion des sessions [Greenberg, 2002]. Nous rapprochons ici cela au fait que l'auteur ait créé trois fils de discussion différents suite à son message initial.
- La nécessité du contrôle de la concurrence dans ce genre de projet collectif et l'importance de sa maitrise. En effet si plusieurs personnes éditent un patch au même moment, leur travail risque d'être perdu dans le cas où quelqu'un poste une version plus récente du patch avant eux ; ou bien générer une incohérence dans la linéarité de l'historique partagé constitué par les différents messages postés.
- La gestion des conflits ; qui dans ce cas ne repose sur aucun support logiciel, elle est gérée localement et manuellement par les utilisateurs, qui doivent alors comparer les versions pour effectuer eux-mêmes la fusion, opération qui n'est pas toujours évidente à gérer.

Si l'expérience apparaît globalement comme plutôt concluante, elle montre donc aussi certaines limites. Un des membres du forum, qui revient *a posteriori* sur ces différentes initiatives, a alors reproché à ce support particulier son caractère linéaire qui empêcherait de donner une direction différente au patch. Il semble aussi souligner la nécessité d'avoir des outils spécifiques pour le patching collaboratif en ajoutant que le forum n'est peut-être pas l'outil le plus adapté à l'heure actuelle pour le pratiquer. Il donne enfin des premières perspectives pouvant améliorer le système :

> MARK DURHAM: *[...] À mesure que l'on travaillait sur les patchs Hippies, la composition prenait peu à peu forme et, inévitablement, se dirigeait dans une direction particulière. Je trouvais à l'époque que la façon dont les messages du forum étaient mis en place n'était en réalité pas très propice à ce genre de chose. Il ne s'agit que d'une seule ligne de haut en bas, sans aucun moyen de diviser ou de bifurquer un sujet – bon pour parler, mais peut-être pas aussi bon pour le développement de patch en groupe.*
> *Ces patchs particuliers sont devenus assez confus, et probablement pas au goût de certaines personnes. Mais imaginez si le fil était plus arrangé graphiquement, avec un arbre comme structure. Les utilisateurs auraient pu se diviser en groupes et faire évoluer le patch loin du patch d'origine, donnant à la composition des directions différentes [...]* [45].

Ce membre parle ici de pouvoir se dissocier du document initial pour pouvoir donner une autre direction au projet, tout en maintenant un lien historique avec ce dernier. Cette structure d'historique se dessinerait alors non-plus linéairement, mais sous la forme d'un arbre à partir duquel on pourrait tirer des branches pour effectuer des modifications de différente nature et de

---

[45] *In* [Durham, 2012], traduit de l'anglais : « *MARK DURHAM:* […] while working on the Hippie patches the composition was slowly taking shape and as is inevitable, heading in a particular direction. I thought at the time that the way the forum posts are set up is actually not very conducive to this type of thing. They're just a single line from top to bottom, with no way for a topic to split or branch off – good for talking, but perhaps not so good for group patch development.

Those particular patches became quite noisy, and probably not to some people's taste. But imagine if the thread was more graphically arranged, with a more tree like structurre [sic]. Users could have branched off in groups and developed away from the whole so the patch, with the composition going in different directions […] »

manière parallèle. La structure décrite ici se révèle en fait très proche des solutions mises en place par les développeurs qui sont amenés à gérer des problèmes d'organisation et de conflits similaires lors de l'activité de conception d'un logiciel à travers l'édition collaborative du même code source. Pour les aider dans cette tâche, ils ont la plupart du temps recours à des logiciels tiers permettant le contrôle de version au sein d'un projet.

2.1.2. Le contrôle de version

Les systèmes de gestion de version[46] sont des outils logiciels conçus pour permettre le tracé des modifications successives apportées au contenu d'un ou plusieurs fichiers, ils comportent généralement aussi des fonctionnalités telles qu'un système d'étiquettes qui permettent de rendre explicites les changements majeurs apportés au sein du projet. Ces outils peuvent être qualifiés de systèmes collaboratifs dans la mesure où ils proposent des solutions qui permettent d'orchestrer le développement parallèle entre plusieurs personnes sur différents fichiers. Cela peut passer notamment par la création de branches, qui permettent d'isoler temporairement une série de modifications, et qui peuvent ensuite être intégrées au projet principal de manière sécurisée grâce à une gestion contrôlée de la concurrence. Leur utilisation représente donc en quelque sorte un premier raffinement par rapport à la méthode de collaboration un peu naïve mise en place au sein du forum en offrant un support logiciel et une meilleure organisation et gestion de l'historique. Après en avoir décrit le principe général, nous verrons dans quelle mesure l'activité d'édition et d'élaboration collective de patch peut bénéficier du support logiciel qu'ils offrent, mais aussi quelles sont leurs limites.

[46] Que l'on appelle *version control system* ou *revision control system* en anglais.

2.1.2.1. Définition

On distingue généralement deux types de gestionnaires de version. Les gestionnaires de version dits centralisés, tels que Subversion[47] (SVN) ; et les gestionnaires de version décentralisés, tels que *git*[48]. La différence majeure que nous noterons entre ces deux types de systèmes réside principalement dans le fait que dans le premier cas, toutes les opérations s'effectuent directement en ligne sur un serveur central – l'utilisateur est alors dépendant du réseau pour réaliser les opérations permises par le système ; les solutions décentralisées au contraire, permettent à toutes les opérations d'être d'effectuées localement et donc d'être exécutées plus rapidement. Nous ne pouvons pas entrer dans plus de détails concernant les spécificités de chacune de ces solutions dans le cadre de cette thèse[49], aussi nous nous concentrerons sur le système *git*, qui est aujourd'hui un outil majoritairement adopté et qui nous est plus familier.

Une personne, désireuse de mettre en place un contrôle de version sur un projet, crée d'abord un dépôt local, l'initialise puis y dépose des fichiers. Elle peut alors commencer à travailler en modifiant le contenu de ces fichiers, puis en publiant les modifications au sein de son dépôt par une opération de *commit*. Chaque publication de modification peut être agrémentée d'un texte court qui décrit la modification apportée au projet, et un historique des modifications peut ainsi être visualisé clairement. Mais l'intérêt principal du système réside surtout dans le fait qu'un utilisateur peut aussi créer un second dépôt, en ligne cette fois-ci, pour y synchroniser sa version locale, et pour permettre à d'autres personnes d'y accéder en parallèle grâce à une opération de clonage du dépôt distant sur leur propre machine. Chaque utilisateur disposant d'une copie

---

[47] https://subversion.apache.org/.

[48] https://git-scm.com/.

[49] Le lecteur pourra se référer à des sources telles que https://git.wiki.kernel.org/index.php/GitSvnComparsion, qui dressent un comparatif beaucoup plus détaillé de ces différents outils.

locale des fichiers peut désormais y apporter les modifications qu'il veut, puis les synchroniser de nouveau avec le dépôt en ligne qui sert de référentiel commun, pour les rendre visibles aux autres collaborateurs. Au moment de synchroniser les données, le système vérifie qu'aucun fichier n'a été modifié parallèlement par un autre utilisateur en effectuant une comparaison entre les données locales et les données distantes. Pour cela il utilise un algorithme de *diff*, [Hunt & McIlroy, 1976], qui permet de générer une représentation compacte des différences entre les deux versions. Ces différences correspondent, au niveau du contenu des fichiers textuels, à l'ajout, la suppression, ou encore la modification de certaines lignes[50]. Ces changements sont ensuite appliqués à la version courante par un mécanisme de *fusion* [Chacon, 2009], et la modification peut être poussée en ligne pour produire une nouvelle version que les autres collaborateurs pourront ensuite récupérer. Dans la majeure partie des cas, les changements sont fusionnés sans problèmes car ils ne se superposent pas à ceux déjà publiés. Mais il arrive parfois que des modifications soient apportées sur les mêmes parties des fichiers de manière concurrentielle – par exemple quand deux utilisateurs modifient la même ligne d'un texte. Dans ce cas, le système ne prend pas de décision, et demande simplement à l'utilisateur de régler la situation lui-même. Il suspend alors temporairement la fusion et ajoute au contenu du document courant un texte appelé *bloc de conflit* qui indique les endroits qui posent problème au sein du fichier. Une fois les conflits réglés, la fusion peut être opérée, les modifications publiées et poussées sur le dépôt distant, et toutes les copies locales sont à nouveau synchronisées.

Les outils de gestion de version sont donc des outils très puissants qui permettent à plusieurs personnes d'effectuer des modifications parallèles sur un même ensemble de fichiers et de

[50] Un exemple de cette représentation est donné dans la sous-section suivante, [Figure 9].

maintenir un historique clair des modifications apportées par chacun des contributeurs au sein d'un projet. Très utilisés par les développeurs informatiques pour l'édition de code source, ces outils peuvent aussi supporter beaucoup d'autres types de fichiers, à partir du moment où il est possible de générer un *diff* de leur contenu. Les documents Max et Pure Data sont tous deux représentés sous forme textuelle – dans un format spécifique dans le cas de Pure Data[51] et au format JSON[52] dans le cas de Max – ce qui les rend donc compatibles avec l'utilisation d'un logiciel de contrôle de version. Il nous faut souligner ici que ces logiciels de gestion de version ont aussi été popularisés ces dernières années par d'autres types d'outils complémentaires qui permettent d'orchestrer la communication, la coordination entre les contributeurs, le suivi de problèmes, ou la gestion plus générale du projet. C'est le cas de plateformes en ligne telles que *Github*[53] ou *GitLab*[54], qui offrent une interface pour gérer des projets versionnés avec *git*, et proposent l'hébergement de dépôt en ligne. Ces services permettent aussi la mise en place d'un contrôle d'accès aux dépôts, à certaines personnes ou strictement privé, et ont souvent une politique incitative favorisant l'open source et le partage publique des projets. On trouve donc aujourd'hui énormément de projets fondés sur Max ou Pure Data qui utilisent le système de gestion de version *git* couplé à ces services en ligne[55]. Nous en avons d'ailleurs nous-même fait usage lors du processus de développement de la bibliothèque HOA[56]. Le dépôt comprenait alors du code source, des ressources, mais aussi un ensemble de fichiers d'aide, d'exemples et de tutoriels, sous la forme de patchs Max ou Pure Data que nous devions produire et modifier à

---

[51] Chaque ligne du fichier correspond à un message valide compris par Pure Data (suivant le protocole FUDI).

[52] JavaScript Object Notation.

[53] https://github.com/.

[54] https://about.gitlab.com/.

[55] Le lien suivant pointe vers certains de ces projets :
https://github.com/search?l=Pure+Data&q=Pure+Data&type=Repositories

[56] https://github.com/CICM/

plusieurs. Si ces outils nous ont été indispensables à l'organisation du projet, et nous ont permis de coordonner le développement collaboratif de manière générale, nous avons aussi pu éprouver à l'époque certaines limitations du système, plus particulièrement lorsqu'il s'agissait de l'édition concurrente d'un même patch.

2.1.2.2. Limites du système

Comme nous étions plusieurs à travailler sur le projet HOA, il est souvent arrivé que nous fassions des modifications locales en parallèle sur les différents patchs du dépôt. Lors de l'opération de synchronisation des fichiers locaux avec les fichiers distants, nous rencontrions alors fréquemment des situations de *conflits de fusion* qui survenaient, comme nous l'avons dit, dès lors que le système n'était pas en mesure de gérer lui-même la *fusion* de manière automatique. Les stratégies généralement adoptées entre les utilisateurs pour gérer ces cas de concurrence au sein d'un projet – et que nous adoptons aussi dans notre propre pratique – sont tout d'abord de les éviter. Pour cela, les utilisateurs peuvent se coordonner eux-mêmes en se fixant certaines règles comme de travailler chacun de leur côté, sur des documents différents. Mais ces solutions temporaires ne nous aident pas à spécifier la nature des problèmes rencontrés. Aussi, observons-les de manière plus concrète, à travers un exemple reproductible assez simpliste, mais déjà représentatif :

Pour cela imaginons une situation où un administrateur crée un nouveau dépôt en ligne, versionné à l'aide du système *git*. Il y dépose un patch vierge, effectue quelques modifications qu'il publie sur la branche principale, puis demande à deux collaborateurs de se présenter en ajoutant un commentaire au patch sur deux branches différentes.

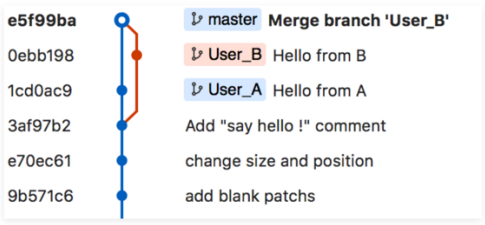

Figure 8 – Historique *Git* qui représente, de bas en haut, trois modifications successives réalisées sur une branche principale (*master*), puis deux modifications effectuées en parallèle par deux utilisateurs sur deux autres branches (*User_A* et *User_B*), puis leur intégration à la branche principale.

Leurs modifications respectives sont alors publiées puis seront fusionnées chacune dans la branche principale pour produire une nouvelle version du patch comportant leurs deux messages. La figure ci-dessus offre une représentation possible de l'historique généré par *git* pour cet exemple [Figure 8].

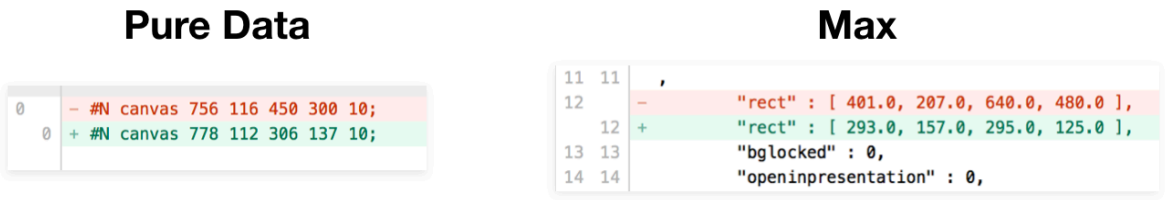

Figure 9 – Représentation d'un *diff* de document Pure Data et Max effectué par le système *git* suite à une opération de repositionnement et de redimensionnement de la fenêtre contenant le patch.

Le premier problème que cet exemple nous permet de pointer est que les informations liées à la vue du patch, notamment la taille et la position de la fenêtre, sont contenues au sein du document de manière globale, que ce soit dans le cas des patchs Pure Data ou Max. Cela a pour nous deux implications principales, la première, liée au système de gestion de version, est que le simple fait de déplacer la fenêtre d'un patch provoque aussi une modification du document qui doit nécessairement être publiée ou annulée dès lors que le patch est sauvegardé par l'utilisateur. Le *diff* généré par une action de ce type est représenté à la [Figure 9]. Outre le fait

que cela génère du bruit inutile dans les modifications faites au document et à l'historique général du projet, cela implique aussi qu'une modification de cet ordre aura aussi un impact sur les autres collaborateurs qui devront alors subir les nouveaux réglages de vue. Si le stockage de ces informations au sein du document nous parait être une solution tout à fait pertinente dans un contexte mono-utilisateur, il nous semble aussi nécessaire qu'il faille que ces informations soient spécifiques et relatives à chaque utilisateur dans un contexte collaboratif et donc d'avoir un système plus flexible.

Le second problème que cet exemple nous permet de mettre en lumière est celui rencontré lors de la *fusion* des modifications apportées parallèlement par les deux utilisateurs à partir de la même version initiale du patch ; il est illustré conjointement par la [Figure 8] et la [Figure 10].

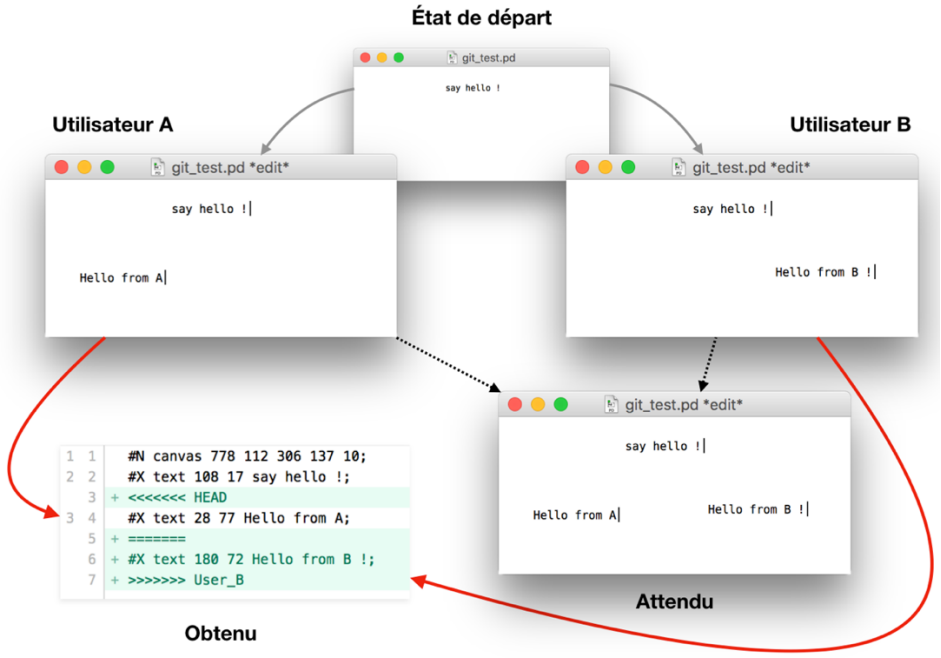

Figure 10 – Schéma représentant le résultat attendu en comparaison de celui obtenu lors d'une tentative de fusion entre deux versions du même document, édités parallèlement par deux utilisateurs et versionnés via le système *git*. Le résultat attendu aurait été que les deux objets ajoutés se retrouvent chacun à leur place au sein du patch de référence. Le résultat obtenu est un document comportant un *bloc conflit* généré par *git* et indiquant que la ligne 3 (correspondant à l'ajout du commentaire dans le patch par l'utilisateur B) existe déjà au sein du document.

L'utilisateur A ajoute un commentaire au patch pour se présenter, il publie cette modification sur sa propre branche, puis la fusionne avec la branche principale. Cette opération s'effectue alors sans problème puisqu'elle consiste, dans le cas de Pure Data, à ajouter simplement une nouvelle ligne au document initial. L'utilisateur B effectue alors les mêmes opérations en parallèle en ajoutant son propre objet commentaire. Le résultat attendu, dans le cadre de cet exemple précis, aurait été que les deux objets soient ajoutés au patch. Mais au moment de fusionner ses modifications à la branche principale, l'utilisateur est confronté à un conflit de fusion dû au fait que la même ligne du document ait été modifiée concurremment [Figure 10]. Le système génère donc un fichier intermédiaire comportant des indications de conflit que l'utilisateur doit alors régler s'il veut synchroniser ses modifications avec celles effectuées par un autre collaborateur. Les logiciels Max et Pure Data ne prennent pas en charge les patchs comportant des blocs de conflits. Aussi, ces marqueurs viennent corrompre le document qui ne peux plus être chargé au sein du logiciel. L'utilisateur, qui doit déjà se résoudre à gérer les conflits d'édition manuellement, doit alors le faire via une représentation textuelle du document avec laquelle il n'est pas du tout familier puisque sa propre représentation du patch est d'habitude visuelle. L'exemple donné ici est plutôt trivial – il lui suffirait seulement de retirer les blocs de conflits pour que les deux objets soient ajoutés au patch – en revanche, on imagine bien que dans des situations plus complexes cette représentation des conflits puisse être plus délicate à interpréter. Dans la majeure partie des cas, il ne s'embêtera donc pas à effectuer de fusion complexe, et préfèrera tout simplement garder l'une ou l'autre version, générant alors le plus souvent une perte du travail rejeté. Notons néanmoins que ce problème spécifique de représentation pourrait théoriquement être réglé si les logiciels de patching pouvaient interpréter et représenter graphiquement les indicateurs de conflits au sein de l'interface graphique du patch, de façon à en proposer une résolution visuelle aux utilisateurs ; néanmoins, aucune solution de ce type n'a encore été mise en œuvre à notre connaissance.

Nous venons de présenter l'apparition de conflits résultant d'un accès concurrent à une même ligne au sein d'un document. Si ces conflits ne peuvent pas être gérés automatiquement par le système, ils ont au moins l'avantage d'être détectés. Les utilisateurs ont donc conscience de la concurrence et peuvent prendre des mesures pour résoudre ces conflits. Aussi le dernier point que nous souhaitons aborder est celui des conflits invisibles, c'est-à-dire non détectés par le système mais qui rendent néanmoins incohérent le document du point de vue du modèle d'application spécifique qu'est le patch. Au sein de l'environnement Max par exemple, chaque objet est stocké au sein du document avec un identifiant unique (par exemple *obj-4*), l'application s'en sert alors pour référencer les objets à différents endroits du document, notamment pour modéliser les *patchlines* – les liens entre les objets, en stockant les identifiants de l'objet expéditeur et destinataire. L'unicité de cet identifiant est garantie localement dans la mesure où il est incrémenté à la création de chaque nouvel objet créé dans le patch. Néanmoins dans un contexte d'édition collaborative, l'unicité de ces identifiants n'est plus garantie et des conflits, non reportés comme tel par le système, peuvent survenir lors d'une édition concurrente. Suite à une fusion réussie, on peut par exemple arriver à des situations ou un lien devient orphelin (relié à un objet absent du document ou à un mauvais port d'entrée ou de sortie), ou pire, relié à un objet autre que celui auquel il avait été initialement et intentionnellement relié, ce qui peut modifier le comportement du patch sans que s'en aperçoivent les collaborateurs. Pour résumer, nous dirons donc que dans la mesure où le système de gestion de version ne connaît pas le modèle de domaine spécifique de l'application auquel s'applique le texte utilisé pour représenter le document - le concept de lien ou d'objet par exemple – il est donc inopérant pour détecter les incohérences d'ordre sémantique présentes au sein du patch.

2.1.3. Bilan de l'étude des solutions de collaboration asynchrone

Nous avons étudié dans cette section deux cas de conception collaborative de patch, exercée de manière asynchrone. Le premier a permis de mettre en évidence l'intérêt musical, ludique et pédagogique de cette approche, mais elle a aussi montré ses limites en matière de pratiques, ainsi que le besoin de recourir à des outils spécifiques destinés à la soutenir. Nous avons ensuite évoqué une autre solution, celle de la gestion de version, qui permet d'avoir un contrôle plus fin pour organiser et orchestrer le travail collaboratif. Nous avons aussi montré que cette solution, appliquée à la collaboration resserrée au sein des documents Max et Pure Data, n'était que partiellement satisfaisante dans la mesure où elle était encore génératrice de conflits.

Si l'on prend le problème sous un autre angle, on se rend compte que dans les deux cas étudiés, les conflits surviennent principalement du fait que les utilisateurs n'ont pas conscience du travail réalisé par les autres parallèlement au leur, puisqu'ils ne le voient qu'après un certain temps et n'en sont pas forcément notifiés tout de suite. Aussi une solution qui permette de transmettre les changements beaucoup plus rapidement aux autres apparaît comme une issue presque logique au problème de la collaboration asynchrone tel que décrit ici. Dans une solution de patching collaborative en temps réel telle que l'envisageons, nous soutenons que le cas de conflit présenté à la [Figure 10] aurait par exemple pu être évité facilement si l'information avait été transmise instantanément à l'autre participant ; la modification apportée par le second utilisateur aurait alors pu être correctement ajoutée à la première plutôt que de s'y substituer.

D'autre part, ces solutions ne permettent pas le partage d'un même espace de travail, donc une interaction plus spontanée entre les utilisateurs. Ils ne voient pas ce que les autres sont en train de faire, ni sur quoi ils travaillent précisément, et répondent encore moins à la problématique

du jeu collaboratif. Aussi, il nous paraît essentiel d'étudier maintenant les principaux outils qui permettent la mise en place d'une collaboration synchrone dans le domaine du patching audio.

2.2. Édition et jeu collaboratifs synchrones

Nous nous situons ici dans les configurations décrites par la première colonne de la matrice d'Ellis, avec des types d'interaction géographiquement distribuées ou colocalisées mais se situant toutes au « même moment » [Tableau 1]. Pour Ellis et Gibbs, les systèmes multi-utilisateurs en temps réel se définissent comme des systèmes où « les actions d'un utilisateur doivent être rapidement propagées aux autres utilisateurs »[57]. Selon si on se situe dans une configuration d'édition ou de contrôle du patch, les actions et donc les informations partagées, ne seront pas les mêmes. De la même manière, si les interactions s'effectuent au même endroit ou à des endroits différents, la façon de mettre en place ce partage aura aussi son importance. Les applications de patching traditionnelles sont conçues à la base pour être utilisées dans un contexte mono-utilisateur ; des adaptations doivent donc être faites pour permettre de les utiliser dans un contexte multi-utilisateur de manière synchrone. Il s'agira alors de faire ressortir ici les choix de conception de ces différents outils, de comprendre quels types de pratiques ils permettent de soutenir et quelles sont leurs limites.

On distingue deux manières de propager les actions aux autres participants et de partager de l'information :

- Le partage d'application. Il permet d'utiliser une seule et même application à plusieurs en même temps. Le traitement de l'information est alors centralisé sur l'application qui

[57] [Ellis & Gibbs, 1989], p. 399., traduit de l'anglais : « Real-time groupware systems are multi-user systems where the actions of one user must quickly be propagated to the other users ».

sert de lien entre les clients qui communiquent avec elle. Dans ce type d'architecture on distinguera aussi des choix de conception qui relève du *partage* et la *réplique* de la vue.

- Le partage de données. Il permet à chaque participant de disposer de l'application sur sa propre machine. Les données doivent alors être partagées et synchronisées entre chaque application client.

Nous présenterons dans un premier temps des solutions existantes permettant la mise en place d'une édition collaborative de patch, puis des solutions dédiées au jeu collaboratif.

2.2.1. Édition collaborative synchrone

Les deux solutions logicielles que nous évoquerons ici sont fondées sur un partage de l'application Pure Data qui permet de la rendre compatible, dans une certaine mesure, avec un contexte d'édition multi-utilisateur. Aussi, avant de les étudier plus en détail, il nous faut faire un point sur l'architecture de cette dernière.

L'application Pure Data est fondée sur une architecture en réseau qui fait communiquer deux processus distincts entre eux via le protocole TCP/IP[58] [Puckette, 1996, p. 3]. Le premier processus, souvent nommé *Pd-engine* ou *Pd-DSP*, s'occupe entre autres de la gestion des documents, du traitement des messages et du signal ; c'est la partie logique de l'application.

[58] *Transmission Control Protocol* et *Internet Protocol*.

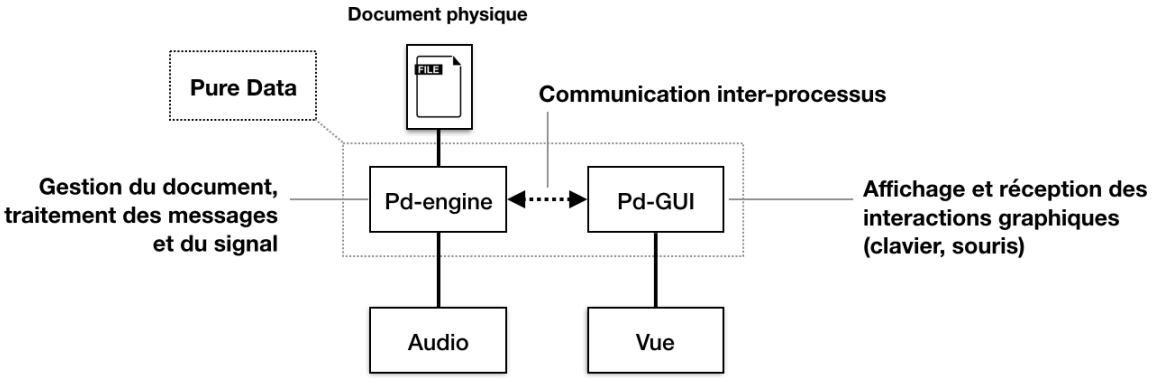

Figure 11 – Schéma présentant une vue simplifiée de l'architecture du logiciel Pure Data et de ses deux processus distincts (*Pd-engine* et *Pd-GUI*).

Le second processus, appelé *Pd-GUI* représente la couche de rendu graphique de l'application. Il communique avec le processus *Pd-engine* par l'intermédiaire de messages. Cette architecture est illustrée par la [Figure 11]. Par défaut les deux processus sont lancés lorsque l'on démarre l'application, mais il est aussi possible de n'en lancer qu'un seul ou encore plusieurs séparément. Les outils que nous nous proposons d'étudier exploitent cette architecture en réseau. Le premier, *peerdata*, est destiné à une édition colocalisée, le second, *PuréeData*, se propose de répondre à la problématique de l'édition synchrone géographiquement distribuée entre les participants.

2.2.1.1. Edition colocalisée

Le projet *peerdata*[59], initialement nommé *multipd*, a été développé par IOhannes M. Zmölnig, dans le cadre du collectif d'art multimédia autrichien centré autour du logiciel Pure Data, intitulé Pd~Graz et fondé en 2005[60]. Ce projet a été utilisé dans le cadre de plusieurs

[59] Le code ainsi que des informations complémentaires sont consultables sur la page relative au projet : https://github.com/umlaeute/peerdata [consulté le 31/08/2018].

[60] https://pd-graz.mur.at/ [consulté le 31/08/2018].

représentations musicales, en particulier lors de la série de performances intitulées « *blind date* », données à la convention Pure Data 2007 à Montréal, ou encore lors de la conférence ICMC en 2008[61] [Pd~Graz, 2009]. Cette suite de représentations entre dans le cadre des pratiques de *Live Coding*, que Zmölnig définit comme : « une performance multimédia, où les interprètes créent et modifient leurs instruments logiciels en temps réel au cours de la performance, par opposition aux performances traditionnelles en informatique musicale, où le contenu pré-produit est lu (musique de bande) ou lorsque les musiciens jouent avec des instruments logiciels conçus à l'avance […] »[62]. Nous sommes donc dans ce cas en présence d'une situation où l'interaction doit s'effectuer de manière synchrone entre des participants tous réunis au même endroit lors de la performance pour éditer le ou les mêmes patchs en vue d'élaborer un traitement de manière collaborative et d'écouter son rendu sonore en temps réel. Quelles sont les solutions techniques qui permettent de soutenir cette pratique ? Zmölnig présente deux types d'architectures permettant la mise en place d'une édition collaborative synchrone de patch Pure Data [Zmölnig, 2007]. Nous les reprenons ici pour essayer d'en évaluer la viabilité.

[61] *International Computer Music Conference*. Un extrait vidéo de la performance est accessible à l'adresse suivante : https://vimeo.com/1643757 [consulté le 31/08/2018].

[62] [Zmölnig, 2007], p.1., traduit de l'anglais : « […] a media performance, where performers create and modify their software-based instruments in real time during the performance, as opposed to traditional computer music performances, where pre-produced content is played back (tape music) or the musicians are performing with ready-made software-instruments […] »

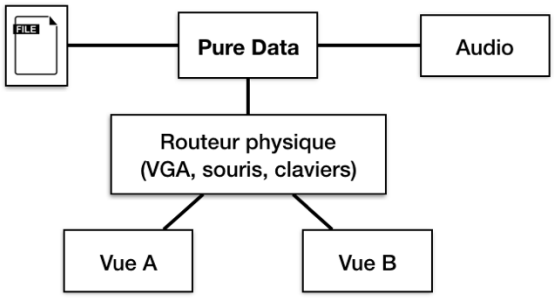

Figure 12 – Schéma présentant un type d'architecture permettant le contrôle d'une même application Pure Data par plusieurs utilisateurs grâce à un routage physique des entrées et sorties. Reprise et traduite depuis [Zmölnig, 2007, p. 2.].

La première, plutôt d'ordre matérielle que logicielle, consiste à utiliser un moniteur, une souris et un clavier par client, puis de les router physiquement vers l'application Pure Data pour permettre à plusieurs personnes de visualiser et d'éditer le même patch simultanément [Figure 12]. Cette première solution – qui s'apparente simplement à un partage de vue, c'est-à-dire à une recopie d'écran avec partage du contrôle – souffre de nombreux problèmes pratiques, notamment le fait que tous les participants sont forcés de visualiser et d'agir sur les mêmes patchs. Lorsqu'un participant effectue une action sur une fenêtre telle que son déplacement, une mise au premier plan ou encore lorsqu'il la fait défiler, cette action est automatiquement répercutée chez les autres participants. Cette solution est donc très limitative, en particulier si le patch est grand et si les utilisateurs veulent travailler sur des patchs différents ou différentes parties du même patch.

L'auteur propose alors une seconde solution qui tire parti de l'architecture en réseau de l'application Pure Data. Celle-ci consiste à répliquer la partie liée au rendu graphique de l'application, c'est-à-dire à exécuter chez chaque client une instance différente du processus *Pd-GUI*.

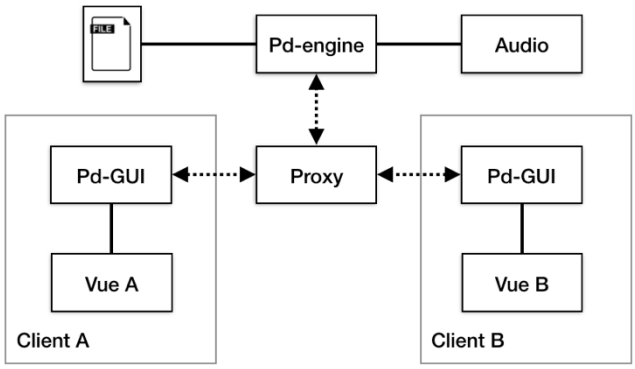

Figure 13 – Schéma présentant l'architecture du projet *peerdata*, repris et adapté depuis [Zmölnig, 2007, p. 2-3]

Ces différentes instances communiquent avec le même processus *Pd-engine*, par l'intermédiaire d'un serveur de type *proxy*[63] qui route simplement les messages entre les deux types de processus [Figure 13]. Les informations reçues de la part de chaque instance de *Pd-GUI* (*e.g.* mise à jour de la position et des actions de la souris, frappes au clavier) sont transmises de manière anonyme au moteur de Pure Data qui les traite comme si elles venaient d'un seul et même utilisateur. Les informations transmises en retour par le processus *Pd-engine* (qui se résument généralement à des instructions de dessin) sont interceptées par le *proxy* qui les distribue alors à chacun des clients de façon à ce que leur vue se mette à jour. Cette architecture permet de résoudre les problèmes liés à la première solution évoquée plus haut [Figure 12]. Chaque client gagne ainsi son indépendance au niveau de la vue : les fenêtres peuvent être placées où ils le veulent, et ils peuvent visualiser des parties différentes du même patch puisque le rendu graphique y est répliqué et non-plus simplement partagé. Cette solution permet donc de visualiser ce que sont en train de faire les autres participants sur le patch, et permet à chaque participant d'agir collectivement sur le rendu sonore du patch qui est réalisé sur le processus *Pd-engine*, commun à tous les participants.

[63] Dans la solution proposée par Zmölnig, il se trouve que la mise en œuvre de ce proxy est aussi réalisée avec le logiciel Pure Data, mais l'auteur aurait très bien pu la faire avec une autre technologie pour un résultat similaire.

Si cette solution paraît en théorie adaptée à l'édition collaborative synchrone, il nous faut nous attarder sur certains effets relatifs à sa mise en œuvre si nous voulons vraiment l'évaluer. Cette solution, qualifiée d'ailleurs de « naïve » par l'auteur lui-même, souffre selon nous aussi de plusieurs problèmes inhérents à ses choix de conception architecturaux. En effet, si la vue est répliquée chez chaque client, la logique d'exécution de l'application Pure Data, et par extension du ou des différents patchs, est toujours contenue au sein d'un même processus central *Pd-engine*. Les utilisateurs disposent donc de leur propre vue mais opèrent sur un contexte d'exécution commun qui n'est pas prévu à la base pour supporter l'édition concurrente de plusieurs utilisateurs. L'auteur pointe comme un dysfonctionnement du système le cas où un des utilisateurs décide de quitter la session. Dans ce cas le message « *; pd quit* » est transmis par l'un des processus *Pd-GUI* au processus *Pd-engine* et provoque non seulement l'arrêt de l'instance chez le client ayant envoyé le message, mais aussi de l'application en général et donc la déconnexion de tous les clients [Zmölnig, 2007, p. 3.], ce qui se révèle pour le moins problématique. Mais ce cas est loin d'être la seule difficulté liée à ce type d'architecture. En effet, beaucoup d'opérations d'éditions courantes dans une application mono-utilisateur sont mal interprétées dans ce système dans la mesure où elles ne sont pas spécifiques à chaque utilisateur mais s'appliquent à l'ensemble du groupe. Pour illustrer l'inadaptation du système à l'édition concurrente resserrée au sein d'un patch, nous pouvons prendre trois exemples que sont les opérations de copier/coller, l'*undo/redo*, ou encore les sélections, mais il y en a en réalité bien plus.

Attardons-nous en premier lieu sur le cas du copier/coller. Un client A veut dupliquer un groupe d'objets au sein du patch. Pour cela il génère une commande de type *copy*, qui, transmise au processus *Pd-engine*, provoque le placement du groupe d'objets sélectionnés dans le presse-

papier global de l'application. Ce même utilisateur peut ensuite envoyer la commande *paste* qui aura pour effet d'ajouter au patch ayant le focus, le contenu du presse-papier. Tout se passe comme il le souhaite, sauf si, entre les deux opérations, un client B a transmis une seconde commande *copy* à l'application. Dans ce cas le client A se retrouvera à coller un groupe d'objets qu'il n'aura pas lui-même choisi de copier, et ne comprendra alors surement pas pourquoi le système réagit de cette façon.

Le cas de l'annulation d'une action ou de sa restauration, plus connu sous la commande du nom d'*undo/redo*, génère aussi le même type d'incohérences. Dans un contexte mono-utilisateur, il est clair que l'action que cherche à annuler un utilisateur correspond à la dernière opération qui vient d'être effectuée sur le document. Mais dans un contexte collaboratif tel qu'il est mis en œuvre ici, l'historique d'édition du patch est partagé, annuler la dernière action pourrait alors provoquer une action non souhaitée dans la mesure où l'utilisateur n'est pas forcément conscient des modifications apportées parallèlement par les autres utilisateurs au sein du patch. Il se peut donc qu'il annule une action dont il n'est pas à l'origine, perturbant alors la manière dont sont en train d'interagir les autres collaborateurs au sein du document. Ce problème est d'ailleurs bien identifié dans la littérature liée aux applications multi-utilisateurs [Prakash & Knister, 1992]. Dans ce type de systèmes, il se révèle en effet primordial de maintenir un historique des modifications non-pas global mais spécifique pour chaque utilisateur.

Le dernier problème que nous citerons dans ce cadre est celui des sélections des différents éléments au sein du patch. Dans ce type de systèmes il n'est pas possible de différencier sa propre sélection de celle des autres dans la mesure où les éléments sélectionnés sont stockés de manière centrale au sein du processus *Pd-engine*. Outre le fait de ne pas pouvoir clairement identifier les actions des autres participants et donc distinguer les actions qu'ils sont en train

d'effectuer, cela peut aussi mener à des problèmes plus gênants dans la mesure où les sélections ont non-seulement un rôle d'indicateur graphique mais aussi un rôle actif au sein du patch. Lorsqu'un utilisateur envoie une commande de suppression, cela a pour effet de supprimer l'ensemble des éléments sélectionnés au sein du patch. L'action de suppression d'un utilisateur pourra alors avoir pour effet de bord de supprimer par inadvertance l'objet sélectionné par un autre utilisateur au sein du patch, ce qui n'est certainement pas l'effet souhaité par l'utilisateur ayant initié cette action.

Ce projet a été conçu pour répondre à la problématique spécifique de l'interaction synchrone colocalisée dans la mesure où le traitement du signal, et donc le rendu sonore est effectué dans la même pièce que celle dans laquelle se trouvent le public et les différents participants à la session de *Live coding* lors de la performance. S'il apporte une première solution permettant à plusieurs personnes de visualiser le même patch en temps réel, on s'aperçoit aussi qu'il génère de nombreuses incohérences au niveau de l'édition si celle-ci est exercée de manière synchrone et resserrée au sein du patch. Pour résumer, on pourrait dire qu'il faut en fait que les actions des différents participants se fassent de manière séquentielle, c'est-à-dire chacun leur tour, plutôt que de manière concurrentielle dans ce type d'approche pour limiter le risque de conflits d'édition. Il faudrait alors ajouter un agent logique au système qui empêche les utilisateurs de faire des modifications au patch quand un autre est en train d'en faire, en ayant par exemple recours à un mécanisme de verrou. Une telle solution pourrait résoudre quelques-uns des problèmes cités plus-haut mais rendrait aussi la solution non-synchrone et d'autres problèmes apparaîtraient. D'autre part, cette solution se révèle aussi totalement inadaptée aux situations d'édition distribuées géographiquement dans la mesure où, si le processus *Pd-engine* était placé sur un serveur central distant, les participants n'entendraient alors plus le rendu sonore sur

lequel ils agissent collectivement du fait que seule la vue est répliquée et pas la logique de traitement.

2.2.1.2. Édition distribuée

Le projet *PuréeData* se situe dans la même veine que *peerdata,* néanmoins quelques différences existent, notamment technologiques dans la mesure où il se propose de répondre à la problématique de l'édition collaborative synchrone géographiquement distribuée[64]. Développé entre 2010 et 2011 par Ted Hayes et Sofy Yuditskaya, il se définit comme une interface graphique pour le logiciel Pure Data permettant le patching collaboratif en temps réel depuis un navigateur Web : « *PuréeData* utilise le système de messagerie interne de Pure Data couplé avec un script écrit en langage *python* pour permettre à n'importe qui de modifier un patch public partagé, tournant sur un serveur, et d'écouter le résultat en temps réel grâce à la transmission d'un flux audio »[65]. Pour cela il utilise le module *Pyata*[66], qui est un module libre et multi-plateforme écrit en langage *Python*, qui permet d'utiliser le logiciel Pure Data comme une API pour y effectuer des opérations à distance sans forcément avoir recours à l'interface graphique (*Pd-GUI*). La technique utilisée par *PuréeData* consiste alors à exploiter les capacités de patching dynamique du logiciel[67]. Aussi appelée *scripting*, cette technique consiste à modifier un patch *à la volée*, en transmettant des commandes à l'application Pure Data sous

---

[64] Ce projet se rapproche aussi du projet *serendiPd* (https://at.or.at/serendipd/ [consulté le 31/08/2018]) dont nous ne parlons pas ici par manque d'information à son sujet.

[65] Tiré de la page d'accueil du projet et traduit de l'anglais : « Using Pd's internal messaging system and an accompanying python script, Pureé Data allows anyone with a browser to modify a public, shared patch running on a server and listen to the results live over an OGG audio stream. ». https://github.com/t3db0t/PureeData [consulté le 31/08/2018]

[66] https://code.google.com/archive/p/pyata/ [consulté le 31/08/2018].

[67] Cette technique est décrite notamment dans un article en ligne disponible à l'adresse suivante : https://jeraman.info/2009/03/22/how-to-use-pure-data-as-a-api/ [consulté le 31/08/2018].

la forme de messages spécifiques qui suivent le protocole FUDI[68]. Il nous est alors possible de créer des objets, de les éditer, les repositionner, les supprimer à distance par la transmission de messages spécifiques. On peut par exemple envoyer un message « *obj 200 600 print;* » pour ajouter un objet *print* à un patch à la position (200 ; 600). Pour créer un lien entre deux objets, on peut envoyer un message « *connect 2 0 3 0;* » ; cela aura alors pour conséquence de relier la première sortie du deuxième objet contenu dans le patch avec la première entrée du troisième objet.

Ces messages peuvent être envoyés depuis l'interface graphique via les objets *send* au sein d'un patch, mais aussi depuis l'extérieur via les objets *netsend* et *netreceive* de Pure Data qui autorisent le transport des messages via un réseau grâce à une *socket* TCP ou UDP[69].

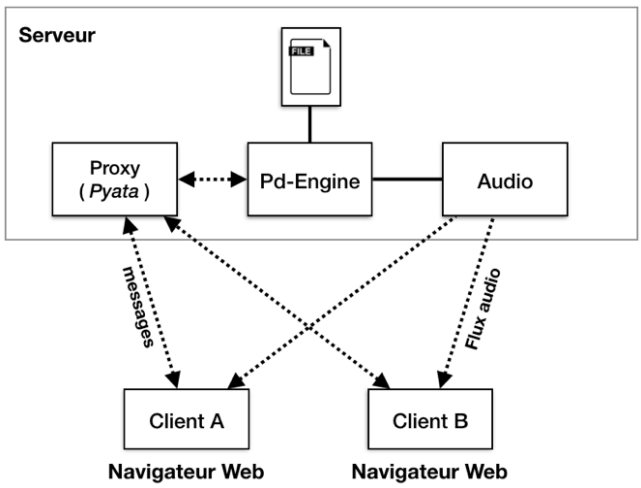

Figure 14 – Schéma présentant l'architecture client-serveur du projet *PuréeData*.

[68] Au 31 août 2018, la seule source que nous avons trouvée relative à ce protocole est accessible à l'adresse suivante https://en.wikipedia.org/wiki/FUDI [consulté le 31/08/2018].

[69] Les acronymes TCP, pour Transmission Control Protocol, et UDP, pour User Datagram Protocol, sont deux protocoles de télécommunication utilisés notamment sur Internet.

C'est cette dernière option qu'ont choisi de mettre en œuvre les auteurs de ce projet. Les clients envoient donc des instructions au patch situé sur le serveur central pour le modifier, et reçoivent en retour des informations qui leur permettent de mettre à jour leur vue. Ce projet présente aussi l'originalité, par rapport à *peerdata* de transmettre l'audio du patch en temps réel aux utilisateurs[70] [Figure 14], ce qui permet *a priori* une édition collaborative synchrone distribuée géographiquement. Mais intéressons-nous maintenant aux limites de ce système.

Le premier point négatif est qu'un seul patch peut être édité à la fois avec cette solution, ce qui le rend en fait très peu flexible. La partie vue au sein de ce projet n'est pas mise en œuvre grâce à une instance de *Pd-GUI*, comme c'était le cas dans *peerdata*, mais à partir d'un navigateur Web. Ceci offre l'avantage de permettre une édition et un contrôle du patch depuis n'importe quel appareil possédant une interface Web, mais l'inconvénient pour les développeurs de devoir recréer toute la partie graphique. De ce fait, et aussi à cause du module *Pyata* utilisé, seulement un petit nombre d'objets natifs de Pure Data y sont supportés[71].

Si on l'analyse de plus près on se rend compte qu'elle souffre en fait des mêmes problèmes de concurrence que dans le cadre du projet *peerdata* étudié plus haut, dans la mesure où elle possède la même architecture d'application partagée avec réplique de la vue. Autrement dit, il n'y a pas de contexte d'édition spécifique propre à chaque utilisateur. L'interface ne peut être qu'identique chez tous les clients dans la mesure où les mêmes informations graphiques sont envoyées à tous les clients sans distinction. D'autre part, tous les participants entendent exactement la même chose dans la mesure où le flux audio est généré une seule fois, côté

[70] Ce projet utilise pour cela un serveur de streaming audio mp3 *IceCast*.

[71] https://github.com/t3db0t/PureeData/wiki/Supported-Objects

serveur, et retransmis en l'état à tous les participants connectés (cela peut ou non être un point négatif selon les cas d'utilisation).

Le dernier problème que ce projet nous permet de mettre en lumière est celui de la réactivité du système. Le délai entre l'action effectuée par un utilisateur et son effet sur sa propre interface graphique est appelé le *temps de réponse* [Ellis & Gibbs, 1989, p. 399]. Dans une architecture dont le traitement est centralisé comme dans celle-ci, le *temps de réponse* est proportionnel à la distance du client par rapport au serveur au sein du réseau. Dans le cadre d'une utilisation où le serveur ainsi que les clients sont géographiquement réunis comme dans le cas d'une performance de *live coding* sur un réseau local, le délai peut être acceptable, mais dans une situation où les clients sont distribués sur plusieurs sites, comme cette solution se propose de soutenir, la latence introduite peut être de plusieurs secondes. Cela rend donc l'interface graphique non-réactive et donc l'édition ou le jeu très difficile dans la mesure où le retour n'est pas du tout immédiat. Cette solution se révèle donc de ce fait inadaptée à l'édition collaborative synchrone, qu'elle soit d'ailleurs colocalisée ou distribuée.

2.2.2. Jeu collaboratif synchrone

L'état de l'art que nous avons dressé jusqu'à présent s'est concentré principalement sur l'activité d'édition collaborative au sein des patchs, mais il nous reste pour être complet à étudier les solutions dédiées à l'activité de jeu. La principale différence avec les solutions précédemment évoquées, qui permettent aussi de modifier les paramètres du patch en temps réel, est qu'on ne s'intéresse plus ici à la manière dont est conçu le traitement mais simplement au contrôle des différents paramètres offerts par l'instrument numérique. Nous étudierons dans un premier temps les solutions permettant la mise en place d'un jeu au sein du même patch lorsque les utilisateurs sont situés physiquement proches les uns des autres. Nous verrons ensuite une dernière solution, *netpd*, qui se propose de répondre à la problématique du jeu

collaboratif synchrone dans un contexte où les participants sont distribués géographiquement sur plusieurs sites.

2.2.2.1. Jeu colocalisé

Les solutions logicielles que nous souhaitons mettre en avant ici permettent le contrôle d'un même patch à plusieurs en temps réel. Elles sont en général conçues pour permettre une interaction entre plusieurs participants réunis au même endroit et connectés au même réseau local en vue d'une performance. Contrôler un patch à distance signifie pouvoir envoyer des messages depuis un appareil vers un autre pour gérer un processus distant. De la même manière et suivant la technologie utilisée, le processus distant peut envoyer des informations en retour à l'interface de contrôle afin qu'elle puisse se mettre à jour et refléter l'état interne du processus contrôlé. On se situe ici non-plus dans le cadre d'un partage d'application mais dans le cadre d'un partage de données entre plusieurs clients connectés à un processus de traitement centralisé. On trouve de nombreuses solutions permettant la mise en place d'un contrôle de patch à distance. Parmi celles-ci on pourra citer l'application *TouchOSC*[72] ou encore *Lemur*[73]. Si ces deux types d'outils permettent de contrôler un même processus distant à plusieurs, ils demandent aussi à l'utilisateur de créer leur propre interface graphique et de mettre en place eux-mêmes la procédure de communication avant de pouvoir commencer à contrôler un patch. Si ces solutions offrent de ce fait une certaine flexibilité à l'utilisateur, elles peuvent aussi représenter selon nous un frein à la collaboration dans la mesure où beaucoup d'étapes sont nécessaires avant de commencer à travailler à plusieurs – configuration manuelle des hôtes et des ports d'entrées et de sorties, mais aussi configuration des messages permettant la

[72] https://hexler.net/software/touchosc [consulté le 20/09/18].

[73] https://liine.net/en/products/lemur/ [consulté le 20/09/18].

synchronisation des différentes interfaces avec le patch (*e.g.* formatage des messages OSC[74] ou MIDI). Aussi nous voudrions citer dans ce cadre une autre solution qui correspond plus à l'idée que nous nous faisons d'un contrôle de patch facilité pour l'utilisateur. Cette solution est la technologie *Mira* développée par Cycling'74[75]. Elle reprend l'idée d'une séparation entre la vue et le moteur du patch. Destiné principalement à un usage sur un réseau local, cet outil permet de contrôler un patch Max grâce à une interface graphique externe située sur la même machine ou sur un autre appareil connecté au même réseau local. L'ajout d'un objet *mira.frame* à un patch crée un canal de communication via le protocole *WebSocket* qui expose automatiquement le contenu du patch à l'extérieur de l'environnement Max. Il est alors possible de s'y connecter, par exemple via un navigateur Web pour être notifié des changements et ainsi contrôler le patch à distance[76]. La technologie *Mira* ressemble donc beaucoup aux autres solutions dédiées au contrôle d'interface en temps réel. La principale différence est que celle-ci bénéficie d'une meilleure intégration au logiciel Max du fait qu'elle ait été développée par les mêmes éditeurs. Par mieux intégrée nous entendons que des ponts ont été réalisés par les développeurs pour faciliter sa prise en main et son usage par les utilisateurs du logiciel. On retiendra de Mira le fait qu'en plus d'offrir un contrôle direct, elle offre aussi un miroir de la vue du patch, c'est-à-dire que l'interface du patch est répliquée à distance sur l'interface de contrôle. Elle propose de plus une procédure de connexion automatique simplifiée qui permet d'éviter la tâche pénible de configuration des paramètres réseau requise par les autres solutions. Cette technologie de contrôle de patch à distance se fond dans une topologie de réseau en étoile. Plusieurs appareils peuvent contrôler un même patch central et un appareil peut contrôler plusieurs patchs différents. L'interface de contrôle n'est en revanche pas autonome, elle dépend

---

[74] *Open Sound Control*

[75] https://github.com/Cycling74/miraweb [consulté le 20/09/18].

[76] L'utilisateur devra alors utiliser la bibliothèque JavaScript suivante prévue à cet effet : https://cycling74.github.io/xebra.js/index.html

du patch central et ne peut fonctionner sans lui. De plus, elle ne permet pas la communication et la réplication du document de patch à patch (par exemple entre deux environnements Max, sur deux machines différentes). L'interface graphique réagit dans le cas de *Mira* aux changements effectués sur le patch et se met à jour automatiquement, néanmoins il est impossible d'éditer le patch depuis l'interface graphique esclave qui sert uniquement à son contrôle.

2.2.2.2. Jeu distribué

Le projet *netpd*, développé depuis 2004 par Roman Haefeli est toujours actif aujourd'hui. Il se définit lui-même comme un CRNMME[77], c'est-à-dire comme un environnement collaboratif en temps réel de création musicale en réseau. Écrite entièrement en Pure Data, cette bibliothèque de patchs permet à plusieurs utilisateurs de se connecter sur le réseau internet pour mettre en place une séance d'improvisation en temps réel avec d'autres participants[78] [Haefeli, 2013]. L'auteur rapproche le type de pratique permise par son système des *Jam sessions* en Jazz. Le but n'est pas ici de faire *sonner* un instrument en l'éditant collectivement comme dans les pratiques de *live coding* évoquées plus haut, mais plutôt de synchroniser plusieurs instances du même instrument sur plusieurs sites distants de façon à ce qu'ils *sonnent* de la même manière chez tous les participants connectés à la session de jeu en réseau.

Le scénario typique d'utilisation de cette solution est le suivant : un groupe de personnes géographiquement distribuées veulent faire de la musique ensemble à travers le logiciel Pure Data. Après qu'ils ont installé l'ensemble des dépendances nécessaires au fonctionnement de *netpd*[79], ils ouvrent un premier patch intitulé *chat.pd*. Ce patch établit une connexion avec le

---

[77] Collaborative Realtime Networked Music Making Environment.

[78] https://www.netpd.org et https://github.com/reduzent/netpd, le lecteur pourra aussi se référer, pour une présentation plus détaillée du projet, à la captation vidéo réalisée lors de la Linux Audio Conference 2013, disponible à l'adresse suivante : http://lac.linuxaudio.org/2013/video.php?id=14

[79] Plusieurs objets externes sont nécessaires à son fonctionnement.

serveur distant puis expose le nom des clients présents ainsi qu'une interface de messagerie instantanée permettant de communiquer avec les autres participants déjà connectés en temps réel. Le lancement d'une deuxième fenêtre, *unpatch*, permet de synchroniser l'ensemble de la session locale de l'utilisateur avec la session globale, c'est-à-dire à la fois les patchs qu'utilisent les autres participants et leur état interne.

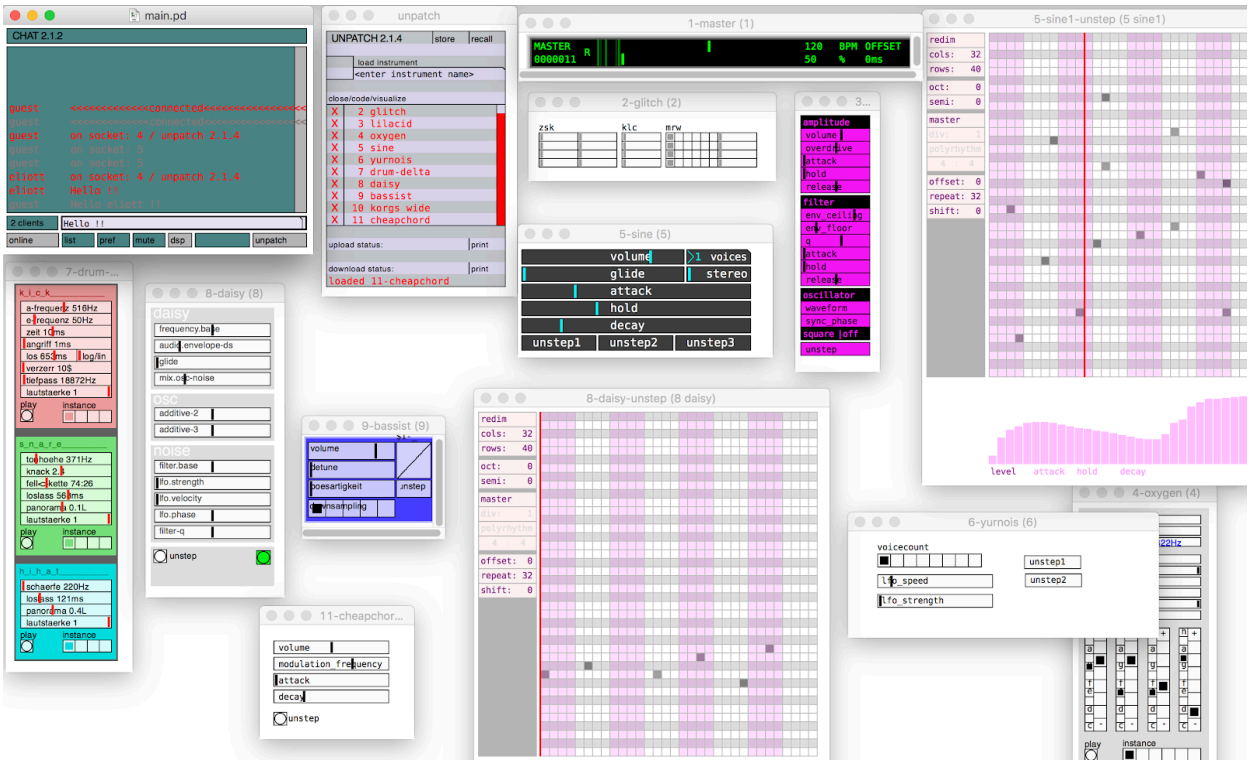

Figure 15 – Capture d'écran exposant un exemple des différentes fenêtres pouvant composer l'interface graphique de Pure Data chez un client utilisant le système *netpd*. On trouve en haut à gauche la fenêtre de messagerie instantanée, à droite de celle-ci une fenêtre nommée *unpatch* qui sert à lister, ajouter ou supprimer les instruments de la session courante, à droite de la fenêtre *unpatch*, le module *Master* qui sert à synchroniser les événements temporels entre les participants ; les autres fenêtres sont des instruments (patchs) dont l'état est synchronisé automatiquement entre tous les participants à la session courante.

L'utilisateur se retrouve alors avec une interface qui peut ressembler à la [Figure 15], à partir de laquelle il lui est possible d'entendre les modifications apportées par les autres participants

en temps réel et d'en apporter lui-même en jouant sur les interfaces de contrôle exposées par les différents instruments de la session.

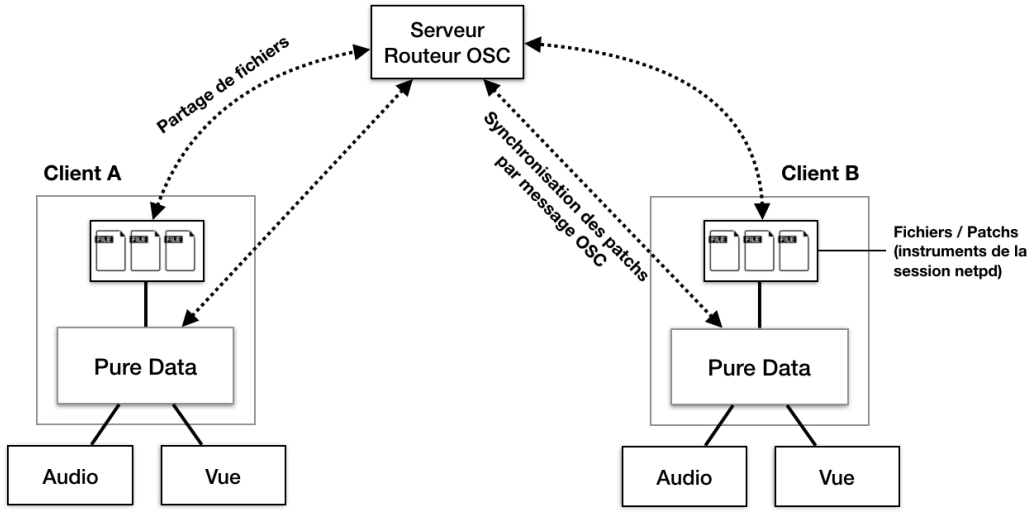

Figure 16 – Schéma présentant une vue simplifiée de l'architecture du projet *netpd* dans une configuration où deux clients sont connectés à une même session de jeu. Chaque client dispose de l'ensemble des documents utilisés lors de la session en local sur leur machine. Ces instruments (patchs) sont exécutés de manière indépendante chez chacun d'eux sur leur propre version de l'application Pure Data. L'état de ces différents instruments est synchronisé par l'intermédiaire d'un serveur central qui sert de pont entre les utilisateurs pour leur permettre de transmettre des informations aux autres participants formatées en OSC.

L'architecture de ce projet change radicalement par rapport à toutes les autres architectures que nous avons pu exposer jusqu'ici dans la mesure où chaque client exécute désormais l'application Pure Data dans son ensemble ; les processus ne sont donc plus partagés mais répliqués entre les clients [Figure 16]. Le traitement (*e.g.* logique, messages, rendu sonore) ne se fait pas de manière unitaire comme dans les autres solutions synchrones que nous avons étudiées mais sur chacune des machines clientes. Le principal intérêt de cette approche est qu'elle répond aux problématiques liés au temps de latence du système et notamment au *temps de réponse* de l'application que nous avons pu mettre en lumière à travers l'étude de la solution *PuréeData*. Toutes les opérations sont effectuées localement sur l'application Pure Data avant d'être transmises aux autres participants. L'interface est donc beaucoup plus réactive aux

changements locaux et se comporte en fait pour l'utilisateur comme si elle était utilisée dans un contexte purement mono-utilisateur. Nous adhérons donc en ce sens totalement à ce choix de conception. Néanmoins, la difficulté dans ce type d'approche se situe désormais dans la manière de synchroniser les données de façon à ce que tous les participants à la session disposent de la même information. Deux choses doivent être synchronisées ici : Les patchs (instruments) destinés à être exécutés lors de la session de jeu ; et l'état de ces différents instruments, c'est-à-dire l'état des différents paramètres de chacun des instruments exécutés. La synchronisation s'effectue dans ce projet exclusivement côté client. Le serveur de *netpd*[80] ne stocke aucun document et ne connaît pas l'état des instruments exécutés. Il a donc simplement pour fonction de permettre la communication entre les différents participants [Figure 16].

Quand un client démarre une nouvelle session, il charge des instruments dont il dispose en local sur sa machine. Lorsqu'un autre client se connecte, le système charge automatiquement les mêmes instruments si celui-ci les détient localement, autrement il les télécharge en les demandant au participant qui les a en sa possession. On est donc dans une situation qui s'apparente à un simple système de partage de fichier. Or les problématiques liées au partage de document, et notamment celui de la gestion des versions sont bien connues. Comment faire pour savoir si la version d'un document est bien la même chez chacun des clients ? La solution mise en place par *netpd* consiste à insérer au sein des différents patchs des *meta-tags* [Haefeli, 2013, p.4.] qui servent notamment à garantir l'unicité des versions. Si un client dispose d'une version trop ancienne d'un instrument il peut donc le mettre à jour en le téléchargeant à nouveau depuis un client qui dispose de la version la plus récente. Ce système marche, mais est selon nous un peu fragile dans la mesure où il se base sur une action de l'utilisateur pour garantir la

[80] Qui se trouve être aussi mis en œuvre sous la forme d'un patch Pure Data, exécuté sur une machine distante accessible par tous les clients.

synchronisation des données. L'utilisateur doit penser à incrémenter à chaque modification la version des instruments s'il veut que la synchronisation s'opère correctement. De plus, si deux utilisateurs font des modifications parallèles au même instrument et incrémentent le numéro de version, le système n'est plus à même de détecter les changements puisque les deux patchs ont des versions égales, et on se retrouve alors avec des problématiques similaires à celles liées au partage de document que nous avons vu émerger lors de notre étude sur la collaboration asynchrone, mais menant en plus ici à des incohérences au niveau de leur exécution. Pour régler ce problème, il faudrait selon nous que ce soit une instance centrale telle que le serveur qui se charge de cette gestion de version en fournissant une version référente des instruments à laquelle tous les participants pourraient se fier.

Les patchs Pure Data ne se synchronisent pas automatiquement avec le système *netpd*. Pour qu'un instrument puisse être utilisé dans le cadre d'une session de jeu collaborative avec *netpd*, ils doivent être conçus ou rendus compatibles avec le système. La bibliothèque *netpd* fournit donc deux choses[81]. La première est une suite de patchs de haut niveau (*netpd-instruments*), conçus spécifiquement pour ce système par les contributeurs du projet – quelques-uns de ces instruments sont présentés à la [Figure 15]. La deuxième est une bibliothèque d'abstraction de bas niveau (*netpd-core*) qui permet de mettre en place la synchronisation des données au sein d'un patch Pure Data standard. Le système *netpd* permet de synchroniser la plupart des types de données qui transitent au sein des patchs à savoir des nombres, des tableaux, des chaînes de caractères ou encore des listes. En étudiant la documentation du projet et la manière dont sont conçus les différents instruments fournis par la bibliothèque, il est donc possible, pour un utilisateur « averti » de Pure Data [Haefeli, 2013, p. 1.], d'adapter ou de créer ses propres

[81] Outre le serveur qui n'a pas vocation à être utilisé directement par les utilisateurs du système.

instruments et de les utiliser de manière collaborative au sein d'une session *netpd* avec d'autres participants. Le choix de synchroniser ou non telle ou telle donnée au sein du patch est laissé au soin du concepteur de l'instrument. Nous adhérons à ce principe qui permet d'offrir plus de flexibilité à la manière dont peut être conçu un processus collaboratif, néanmoins nous sommes aussi conscients qu'il peut représenter une relative complexité pour un utilisateur débutant. La plupart du temps les utilisateurs se serviront des instruments tels que les a développés et conçus le programmeur, mais n'en créeront donc pas de nouveau. Ce processus de création d'instrument pourrait être selon nous facilité si un utilisateur novice dans la pratique du patching et la création d'instruments synchronisés avait la possibilité de collaborer avec un utilisateur plus avancé qui pourrait l'assister dans cette tâche en temps réel. Malheureusement, ce système ne permet pas d'éditer le même patch à plusieurs en temps réel. Cette solution logicielle n'a pas été conçue dans le but de partager l'étape de conception du patch en temps réel et propose donc seulement une approche de haut niveau destinée à la mise en place d'un jeu collaboratif en réseau avec Pure Data comme support. Il représente néanmoins pour nous la solution la plus aboutie de l'état de l'art en matière de collaboration synchrone au sein des environnements de patching, notamment par son architecture permettant le partage des données.

2.3. Bilan de l'étude des pratiques et solutions logicielles existantes

Nous avons présenté dans ce second chapitre des outils logiciels permettant la collaboration dans le cadre des pratiques liées au patching audio. Les solutions abordées ont d'abord été regroupées dans deux principales catégories relatives à la temporalité de l'interaction, telles que définies par la matrice d'Ellis [Ellis & al., 1991], [Tableau 1]. Nous avons présenté deux approches différentes de collaboration asynchrone, celle mise en place à partir du forum comme support puis celle liée à l'utilisation d'un gestionnaire de version. Dans le cas du forum, les membres postaient une nouvelle version en téléversant le document dans son intégralité sur la

plateforme en ligne, qu'ils attachaient à un nouveau message. Il n'était alors pas possible de visualiser clairement les modifications apportées au document, sinon par les indications textuelles apportées par leurs différents commentaires, et des risques de conflits pouvant générer une perte de données par réécriture étaient inhérents au procédé. Il n'y avait donc pour ainsi dire aucun contrôle logiciel de la concurrence, celui-ci étant exercé manuellement par les utilisateurs. Nous avons alors étudié d'autres solutions comme les outils de contrôle de version. Si ces solutions paraissent séduisantes à première vue et permettent la mise en place d'une collaboration au sens large, mieux orchestrée, via notamment des solutions de gestion de projet en ligne, la collaboration ne peut y être exercée que de manière asynchrone, la gestion des conflits repose encore en grande partie sur les utilisateurs, enfin si la synchronisation des patchs est possible, même de manière asynchrones et non automatisée, les utilisateurs ne disposent pas d'un espace commun pour effectuer les opérations à plusieurs en même temps sur le document. Ils ne voient pas ce que sont en train de faire les autres au moment où ils réalisent une action mais uniquement au moment où l'utilisateur a décidé de la partager. Les notifications ne sont pas automatiques, ou alors trop espacées dans le temps pour donner à l'utilisateur le sentiment de travailler en collaboration étroite avec quelqu'un. Dans les deux systèmes étudiés, c'est par exemple à l'utilisateur de se renseigner pour savoir s'il y a eu un changement au sein d'un document édité de manière collaborative. Dans le cas de la collaboration asynchrone l'espace de production est purement local, il correspond à un usage mono-utilisateur des logiciels Max et Pure Data, il n'y a pas de partage de cet espace entre les utilisateurs, en revanche, les utilisateurs partagent le produit de leur espace de production personnel qu'est le patch, en tant que document. Pour les aider dans cette tâche, des solutions telles que le forum ou les logiciels de gestion de version permettent d'assister l'élaboration collaborative du patch en apportant des fonctions de *communication* (*e.g.* commentaires sous la forme de messages sur le forum) ou

encore des fonctions de *coordination* comme les indicateurs de conflits dans le cas des logiciels de gestion de version.

Aussi, il nous a semblé nécessaire d'étudier ensuite les solutions disponibles à l'heure actuelle pour exercer un patching collaboratif en temps réel qui permettent de résoudre, dans une certaine mesure, certains des problèmes inhérents à la collaboration asynchrone grâce à une propagation plus rapide des informations entre les utilisateurs et en fournissant des interfaces graphiques permettant le partage d'un espace de travail commun. Nous avons alors passé en revu une série de projets et d'outils développés pour permettre la mise en place d'une collaboration synchrone à partir des environnements Max et Pure Data. Ces projets ont alors été répartis en fonction de la principale activité qu'ils se proposent de soutenir, à savoir celle de l'édition ou du contrôle du patch, mais aussi en fonction du lieu où peut se dérouler l'interaction entre les utilisateurs (dans le même lieu ou dans des lieux différents). Ces solutions nous ont permis de soulever de manière concrète plusieurs problématiques liées à la collaboration en temps réel appliquée au patch, notamment au niveau des pratiques et des attentes des utilisateurs, mais aussi de certaines difficultés sur le plan technique de la mise en œuvre de ces solutions logicielles.

3. Formalisation des enjeux

Ce chapitre est dédié à la formalisation des principaux enjeux liés à la mise en œuvre d'une solution de patching audio collaborative. Certains de ces enjeux sont communs à ceux que l'on trouve de manière générale dans la conception des systèmes multi-utilisateurs, d'autres demandent à être mis en perspective avec notre propre domaine d'étude. Nous apporterons ici des précisions quant aux différentes notions déjà abordées au cours de l'étude des solutions logicielles existantes pour entrevoir le type de solution que nous avons voulu développer dans

le cadre plus spécifique du projet MUSICOLL. Afin de clarifier l'exposé, nous avons choisi de présenter ces différents enjeux de manière indépendante, mais nous verrons néanmoins qu'ils sont pour la plupart tous interdépendants.

3.1. Caractéristiques des systèmes synchrones

C. A. Ellis et S. J. Gibbs définissent deux notions clefs liés aux systèmes collaboratifs, le *temps de réponse* et le *temps de notifications* [Ellis & Gibbs, 1989, p. 399] :

- Le *temps de réponse* d'un système correspond au temps nécessaire pour que l'action d'un utilisateur soit reflétée sur sa propre vue.
- Le *temps de notification* est le temps nécessaire à ce que l'action d'un utilisateur soit propagée aux autres utilisateurs sur leur propre interface.

Ces auteurs listent ensuite les caractéristiques qu'ils considèrent comme propres aux logiciels collaboratifs en temps réel comme suit :

- **Hautement interactif** - les temps de réponse doivent être courts.
- **Temps réel** - les temps de notification doivent être comparables aux temps de réponse.
- **Distribué** - de manière générale, on ne peut pas supposer que les participants sont tous connectés à la même machine ou même au même réseau local, les solutions ou le traitement est centralisé sur une seule machine sont donc exclus.
- **Volatile** - les participants sont libres d'aller et venir pendant une session[82].
- **Ad hoc** - généralement les participants ne suivent pas de scénario pré-planifié, il n'est donc pas possible de dire *a priori* quelles informations seront accessibles.

[82] La notion de sessions sera abordée à la section 3.4 (p. 92) de ce document.

- **Resserrés** - pendant une session, il y a un haut degré de *conflit d'accès* lorsque les participants travaillent et modifient les mêmes données[83].

Dans le cas d'un environnement de patching audio collaboratif en temps réel, les notions de *temps de réponse* et de *temps de notification* doivent être mis en perspective à la fois au niveau du rendu graphique des éléments du patch mais aussi de son rendu sonore. Un *temps de réponse* trop long au sein d'une interface de contrôle provoquera alors à la fois une latence de l'affichage de la valeur mais aussi une latence de la mise à jour du paramètre sur lequel il doit agir au sein du traitement et donc un manque de réactivité du système encore plus perceptible.

3.2. Réplique des données

On retrouve l'opposition entre les architectures centralisées et les architectures permettant la réplique des données à la fois dans les solutions logicielles permettant la collaboration synchrone et asynchrone. Les deux types d'architecture ont chacune leurs avantages et leurs défauts. Une architecture centralisée est en générale plus facile à mettre en place mais souffre de problèmes de concurrence inhérents ainsi qu'une latence due au fait que les opérations doivent d'abord transiter sur le réseau avant d'être prises en compte localement. D'un autre côté, les solutions décentralisées offrent plus de flexibilité, permettent de travailler plus rapidement en effectuant les opérations sur le document de manière locale et donc de ne pas subir les aléas du réseau ; mais en contrepartie, elles sont souvent plus difficiles à mettre en œuvre dans la mesure où cette architecture implique la mise en place d'un mécanisme de synchronisation des répliques locales qui devra alors assurer la cohérence et la convergence des données sur tous les sites distribués. Elles doivent aussi préserver leur intégrité du point de vue

[83] Cette problématique est celle du contrôle de la concurrence qui sera abordée à la section 3.5 (p. 94) de ce document.

de l'intention des utilisateurs et du modèle d'application [Ellis & Gibbs, 1989], [Sun & al., 1998]. Nous avons vu avec l'étude de la solution *netpd*, qui est la seule solution synchrone mettant en place une architecture permettant la réplique des données que des problèmes pouvaient apparaître si elle était mise en œuvre de manière trop fragile, notamment dans le cas de la synchronisation des différents patchs entre les utilisateurs.

Les projets que nous avons évoqués précédemment exposent des topologies de réseau et des architectures clientes différentes pour la mise en place de la collaboration en temps réel. Alors que des projets tels que *peerdata* et *PuréeData* s'inscrivent clairement dans une architecture de traitement centralisée avec un partage de l'application (dans la mesure où le moteur de Pure Data tourne sur le serveur et pas sur l'appareil où l'utilisateur voit le patch s'afficher), certains comme *netpd* se situent plus dans une approche décentralisée ou chaque utilisateur dispose de sa propre instance de l'application Pure Data. Selon C. A. Ellis, S. J. Gibbs et G. Rein, « chaque approche a ses avantages et ses inconvénients. Alors que la première [*approche centralisée*] permet d'utiliser les applications existantes, chaque utilisateur a une vue identique de l'application - il n'y a pas de contexte par utilisateur. La deuxième approche [*par réplique des données*] offre la possibilité d'une interface plus riche, mais l'application doit être conçue à partir de zéro ou avec un effort supplémentaire considérable »[84]. Dans le cas de *netpd* cet « effort supplémentaire considérable » doit être fourni par le concepteur de l'instrument qui doit rendre son patch compatible avec le système pour qu'il puisse être correctement synchronisé entre tous les participants dans l'application Pure Data.

[84] [Ellis & al., 1991, p. 42-43], traduit de l'anglais : « Each approach has its advantages and disadvantages. While the first allows existing applications to be used, each user has an identical view of the application - there is no per-user context. The second approach offers the possibility of a richer interface, but the application must be built from the ground up or with considerable additional effort. ».

D’autre part, le fait que le traitement (des messages et de l’audio) se fasse au sein d’une seule application dans le cadre des projets mettant en œuvre une architecture centralisée ne permet d’avoir qu’un seul rendu sonore. Si ce choix peut paraître pertinent dans le cadre d’une performance localisée géographiquement comme ce peut être le cas dans des pratiques de type *live coding*, [Pd~Graz, 2009], [Zmölnig, 2007], il l’est beaucoup moins dès lors que les utilisateurs sont éloignés ou cherchent simplement à disposer de leurs propres valeurs de jeu au sein du patch. Nous aurons l’occasion d’aborder à nouveau ce point un peu plus loin dans la section dédiée aux enjeux relatifs à la mise en place d’un contexte spécifique par utilisateur.

3.3. Contrôle d’accès

Le contrôle d’accès fait partie de l’espace de production tel que défini par le trèfle des systèmes multi-utilisateurs [Figure 3, p. 31]. D’un point de vue logiciel, le contrôle d’accès est le fait de pouvoir limiter l’accès à une ressource ou de pouvoir déterminer qui a accès à quoi au sein d’une application, d’un système, ou plus spécifiquement d’un document. Le contrôle d’accès se doit en ce sens d’être explicité par l’interface graphique du logiciel afin que l’utilisateur soit en mesure de savoir s’il peut accéder à une ressource ou non, mais aussi l’impact qu’aura le fait de modifier un élément spécifique au sein de l’espace commun du patch sur les autres utilisateurs.

Dans les solutions asynchrones étudiées, l’utilisateur n’a par exemple pas accès aux données relatives à l’exécution du patch en temps réel, on peut donc dire que les variables de jeu du patch restent locales. Dans le cas d’une architecture centralisée qui permet le partage synchrone de l’espace de production telle que *peerdata* ou *PuréeData*, toutes les données au sein du patch sont publiques (que ce soit celles liées à la structure du document ou des variables liées à l’exécution du patch), au même titre que l’est en fait l’ensemble de l’application. Dans le cas

des solutions de jeu synchrones, seuls certains paramètres de jeu du patch sont partagés et publics. Dans le cas de la technologie *Mira*, les données partagées seront celles que le concepteur du patch aura décidé de rendre publiques (en choisissant quelles interfaces graphiques il rend contrôlable depuis l'extérieur du patch). Dans le cas de *netpd*, seules les données que l'utilisateur aura choisi de synchroniser avec les autres à travers la construction de l'instrument collaboratif seront partagées et donc rendues publiques.

L'enjeu principal du contrôle d'accès et plus généralement de la modélisation de l'espace de production mis en commun se situe donc dans la définition de la frontière entre ce qui est public, partagé au niveau du patch ou de l'application – donc ce qui peut être vu ou entendu par les autres utilisateurs – de ce qui est local, privé – que les autres utilisateurs ne peuvent donc pas voir ni entendre ou qui n'aura pas d'impact sur eux. Autrement dit quelles sont les données que l'on choisit de mettre en commun et de partager ?

3.4. Gestion des sessions

La gestion des sessions dans les logiciels multi-utilisateurs permet, de manière générale, aux personnes de se retrouver virtuellement au sein d'un collecticiel et de contrôler la manière dont ils sont interconnectés, en ce sens elle entre dans les fonctions de *coordination* définies par le modèle conceptuel d'Ellis [Ellis & Wainer, 1994] repris par Salber [Salber, 1995], [Figure 3, p. 31]. Le collecticiel doit alors être en mesure de permettre d'établir des canaux de production et/ou de communication isolés permettant à plusieurs groupes de personnes de travailler simultanément et de manière indépendante sur différents documents ou projets. Ces canaux de communication peuvent être de différentes natures suivant le domaine d'application. Ces derniers sont souvent exprimés sous la forme de métaphores spatiales telles que des « chambres » virtuelles dans lesquels peuvent se « retrouver » plusieurs participants

[Greenberg, 2002]. Dans notre cas ces « chambres » peuvent être des documents de type *patch* au sein desquels plusieurs personnes peuvent se connecter pour se retrouver au sein d'un même espace graphique afin d'élaborer le même traitement sonore ou de le contrôler à plusieurs. Ces sessions peuvent aussi être temporaires ou persistantes. Dans le dernier cas, l'utilisateur pourra reprendre la session en l'état à un autre moment en ouvrant de nouveau le même document. La gestion des sessions est aussi liée à la problématique du contrôle d'accès, c'est-à-dire de déterminer qui a accès à telle session ou document, si cet accès est public et ouvert à tous ou s'il est privé et restreint à une personne ou un groupe de personnes.

Les solutions synchrones évoquées au chapitre précédent mettent en œuvre différemment ce principe mais, de manière générale, elles ne prennent en compte ni la sauvegarde en ligne ni le contrôle d'accès. *PuréeData* ne permet de disposer que d'une seule session qui correspond à un document unique, c'est-à-dire qu'un seul patch ne peut être édité de manière collaborative. Il n'est donc pas possible de choisir sur lequel on veut travailler et on ne peut pas en créer de nouveaux. Dans une solution telle que *peerdata*, la session correspond en fait non-pas à des patchs mais à l'ensemble de l'application du fait de sa mise en œuvre qui impose un partage du moteur de Pure Data. De ce fait, les utilisateurs n'ont d'autres choix que d'être automatiquement connectés à l'ensemble des patchs de l'application ouverts par le groupe. La mise en œuvre actuelle de *netpd* offre une seule session partagée qui correspond à un ensemble de patchs. Cette solution ne permet donc pas de créer de groupes isolés comportant leurs propres instruments. Enfin, la solution Mira semble offrir sur ce point une approche plus souple dans la mesure où elle permet la création de multiples sessions indépendantes de jeu, qu'un groupe d'utilisateurs spécifiques peuvent rejoindre.

La solution idéale en matière de gestion de sessions serait donc à notre sens de permettre à différents groupes d'utilisateurs de créer eux-mêmes leurs propres sessions sous la forme de

documents sauvegardés en ligne, de façon à pouvoir les rouvrir ultérieurement en l'état, les rejoindre ou les quitter à loisir, mais aussi de pouvoir contrôler qui y a accès.

3.5. Gestion des conflits et contrôle de la concurrence

Le contrôle de la concurrence est un aspect crucial dans la conception d'un collecticiel. Il doit être mis en place pour garantir la cohérence des données partagées entre les personnes, prévenir, et gérer les conflits potentiellement générés par les actions parallèles et simultanées des utilisateurs. Pour gérer les cas d'édition concurrente, plusieurs approches sont envisageables, que ce soit dans le cadre d'une pratique collaborative synchrone ou asynchrone. On peut les regrouper de manière générale en deux catégories : les approches *pessimistes* et les approches *optimistes* :

- Dans une approche ***pessimiste*** la collaboration est permise grâce à la mise en place d'un mécanisme de verrou qui permet l'édition des données ou d'un document à tour de rôle. Les approches les plus restrictives limitent l'accès au document entier, il n'est alors pas possible d'effectuer la moindre opération sur un document quand un autre utilisateur est déjà en train d'y travailler. Les approches plus permissives permettent de limiter l'accès seulement à certaines parties du document.
- Dans une approche ***optimiste***, le système n'a pas besoin d'être verrouillé pour laisser les utilisateurs lire ou éditer les données. Tous les utilisateurs peuvent donc les lire sans qu'il n'y ait de blocage. Lors d'une mise à jour des données – suite à une opération d'édition du document par un utilisateur par exemple – le système vérifie si celles-ci ont déjà été modifiées préalablement. Dans ce cas, le logiciel génère une erreur qui peut être résolue manuellement par l'utilisateur ou automatiquement par le système.

C. A Ellis et S. J. Gibbs définissent le contrôle de la concurrence comme « le moyen d'aider à la résolution des conflits entre les participants, et leur permettre de réaliser des actions de groupe resserrées »[85]. Ce resserrement se comprend selon nous à la fois dans sa dimension temporelle, mais aussi au niveau des éléments sur lesquels se portent les actions exercées par le groupe. Un cas typique de conflit qu'un collecticiel synchrone de patching audio devra gérer est par exemple une situation où un utilisateur tente de raccorder un lien à un objet qui est supprimé au même moment par un autre utilisateur. Il s'agira aussi de pouvoir décider ce qui se passe quand deux utilisateurs déplacent le même objet au sein du patch. D'autre part, il faut aussi trouver les moyens de rendre explicites ces conflits pour l'utilisateur. Ces moyens peuvent être compris notamment dans ceux liés au support des mécanismes de conscience de groupe que nous étudierons par la suite.

Aucune des solutions collaboratives de patching en temps réel évoquées au chapitre précédent ne propose de réel contrôle de la concurrence et certaines semblent même ignorer le problème. Les approches collaboratives asynchrones que nous avons citées semblent au contraire le considérer, en revanche, aucune des solutions proposées ne semblent être une réponse efficace au problème. Dans le cas du forum, la gestion de la concurrence est confiée aux utilisateurs grâce à un système pseudo-pessimiste qui consiste à bloquer un espace temporel au sein duquel seul un utilisateur peut apporter des modifications au document. Dans le cas de l'utilisation d'un outil de gestion de version, plusieurs utilisateurs peuvent réaliser des actions parallèles sur des documents différents ou des parties différentes d'un même document sans que cela pose problème, car les modifications peuvent être fusionnées. En revanche plus les modifications

[85] Tiré de [Ellis & Gibbs, 1989], p. 400., traduit de l'anglais : « Concurrency control is needed within groupware systems to help resolve conflicts between participants, and to allow them to perform tightly coupled group activities »

sont resserrées au sein du document, notamment lorsqu'elles portent sur le même élément, moins il y a de chance que le système puisse les gérer. Des cas de conflits arrivent alors fréquemment et c'est aux utilisateurs de les régler manuellement, ce qui rend la solution d'autant plus fragile dans la mesure où rien n'assure que le résultat final obtenu soit celui souhaité par le groupe ou que le document reste valide du point de vue du modèle d'application spécifique du patch.

Dans le cadre d'une application collaborative synchrone, l'utilisateur s'attend à avoir un temps de réponse du système qui soit le plus faible possible, que ce soit au niveau du rendu graphique de l'interface du patch ou de son rendu sonore. Greenberg souligne à ce propos que les méthodes de contrôle de la concurrence pessimistes (par blocage ou verrou) ne satisfont pas ces exigences dans la mesure où l'utilisateur, se situant au cœur du processus, ne tolérera souvent pas la latence qu'elles introduisent [Greenberg, 2002]. Un utilisateur ne devrait pas subir le fait que le document soit édité ou utilisé par plusieurs personnes à la fois. Il s'attend à pouvoir effectuer les mêmes opérations que dans un contexte mono-utilisateur, sans que cela ne génère de conflit qu'il devrait lui-même régler. La gestion de la concurrence doit donc se faire selon nous de la manière la plus transparente qui soit. La solution à adopter semble alors être de suivre une approche optimiste qui consiste à gérer les cas d'édition resserrés au sein du document, et ce de manière automatique pour ne pas laisser cette tâche aux utilisateurs du logiciel.

3.6. Mécanismes de conscience de groupe

Dans un système collaboratif, chaque utilisateur doit avoir conscience des actions des différents participants afin de situer sa propre action dans la perspective du groupe. Le terme *awareness*, que nous traduisons par *conscience*, revient souvent dans la littérature liée au travail collaboratif et coopératif assisté par ordinateur. La notion de conscience de groupe rassemble alors

beaucoup de choses : conscience des changements (*change awareness*) [McCaffrey, 1998], conscience de l'espace partagé (*workspace awareness*) qui peut comprendre aussi la conscience des données partagées, la conscience de la concurrence, de la présence ou encore de la visite des autres utilisateurs [Gutwin, 1997], [Gutwin & Greenberg, 1997], [Gellersen & Schmidt, 2002].

Dans un environnement physique – par opposition à l'environnement virtuel d'un collecticiel – où les personnes sont regroupées dans le « *même lieu* » au « *même moment* » et où les interactions s'effectuent donc en face-à-face, le recours à une communication verbale ou à des gestes qui peuvent venir assister le travail collaboratif est fréquent pour permettre la coordination des actions réalisées par les différents utilisateurs [Tang, 1991]. Ils peuvent par exemple pointer du doigt un élément de l'espace commun pour en parler aux autres collaborateurs avec plus de précision ou encore s'apercevoir qu'un autre utilisateur est en train de réaliser une action spécifique en l'observant simplement. Dans une situation où la collaboration est distribuée, dans le temps ou dans l'espace, ces interactions directes ne sont plus possibles et les participants ne peuvent plus interagir de la même manière. Or le besoin de coordination entre les utilisateurs existe toujours et se révèle même un problème d'autant plus essentiel à considérer. Dans ce type de situation il faut donc trouver des manières de pallier le manque d'information dû à l'absence de contacts physiques entre les participants en apportant des solutions qui permettent de coordonner à nouveau les activités des utilisateurs. Cette coordination peut être assistée notamment par l'interface du logiciel qui devra alors apporter ces clefs de compréhension. Les outils conceptuels liés aux mécanismes de conscience dans les systèmes coopératifs ont été élaborés pour pallier le manque d'information dans le cadre d'un environnement virtuel ou les personnes sont distribuées géographiquement à des endroits différents ou collaborent à des moments différents et ne peuvent donc plus utiliser les

interactions directes en face-à-face qui viennent normalement assister le travail de groupe. Cette conscience peut se porter sur des objets différents suivant le mode d'interaction qui est en jeu.

Dans le cadre des interactions asynchrones on cherchera par exemple à savoir ce qui s'est passé sur un document durant son absence, les utilisateurs doivent alors avoir conscience des changements effectués par les autres collaborateurs [McCaffrey, 1998]. Les solutions asynchrones que nous avons étudiées apportent en ce sens des premières réponses en fournissant par exemple un historique des modifications apportées au document qui prennent la forme de messages sur le forum, d'une coloration spécifique des objets du patch pour pointer la contribution de chaque utilisateur au sein du patch [Figure 5, p. 44], ou encore de messages associés aux *commits* dans le cadre de l'utilisation des outils de gestion de version [Figure 8, p. 60].

Dans un contexte où les interactions s'effectuent de manière synchrone au sein d'un espace partagé, les exigences en matière de conscience peuvent différer dans la mesure où l'utilisateur peut s'apercevoir des changements effectués par les utilisateurs en temps réel directement au sein de l'interface. En revanche, comme plusieurs personnes sont présentes en même temps sur le même espace, il leur faut aussi des moyens de coordonner leurs actions pour assurer une meilleure synergie. C. Gutwin et S. Greenberg appellent *workspace awareness* la connaissance immédiate qu'une personne détient à propos de l'interaction d'une autre personne au sein d'un espace de travail [Gutwin & Greenberg, 1997]. « La conscience de l'espace de travail partagé aide les personnes à passer d'une activité individuelle à une activité partagée, fournit un contexte dans lequel il est possible d'interpréter l'intention des autres personnes et d'anticiper

leurs actions, elle réduit l'effort nécessaire pour coordonner les tâches et les ressources »[86]. Cette conscience comprend alors la possibilité de savoir *qui* est sur l'espace de travail, *où* les personnes sont en train de travailler, *ce* qu'elles sont en train de faire et enfin ce qu'elles ont *l'intention* de faire ensuite.

La mise en place de ces mécanismes est fortement liée aux aspects de conception de l'interface graphique qui devra alors fournir ce type d'informations. Dans les solutions étudiées de patching synchrone, aucun mécanisme de conscience de groupe n'a été mis en place – à part peut-être dans le cadre du projet *netpd* qui propose un système de messagerie instantanée qui permet de répondre à la question du *qui* est actuellement connecté à la session courante. Dans l'ensemble de ces solutions, l'utilisateur ne distingue pas les actions des autres utilisateurs ou bien ne peut les différencier de ses propres actions. On pourrait néanmoins imaginer un système plus adapté qui puisse notifier de la présence des autres utilisateurs au sein du patch, en affichant les noms des participants, la zone qu'ils sont en train de visualiser, ou encore ce qu'ils sont en train de faire. Par exemple, une solution pourrait être de différencier les sélections des utilisateurs distants par rapport à celles de l'utilisateur local. La mise en place de ce type de solution implique alors de pouvoir disposer d'un contexte d'affichage qui puisse s'adapter à chaque utilisateur.

3.7. Contexte spécifique par utilisateur

Une des approches possibles relatives à la construction d'interfaces collaboratives est connu sous le nom de *WYSIWIS* [Stefik & al., 1987]. Cet acronyme de langue anglaise correspond à

[86] Citation tirée de [Gutwin & Greenberg, 1997, p. 1] et traduite de l'anglais : « workspace awareness helps people move between individual and shared activities, provides a context in which to interpret other's utterances, allows anticipation of others' actions, and reduces the effort needed to coordinate tasks and resources. »

« What You See Is What I See », que l'on peut traduire littéralement en français par « Ce que vous voyez est ce que je vois ». Il décrit des interfaces qui permettent le partage d'un contexte graphique qui est garanti comme identique et qui apparaît donc de la même manière chez tous les participants. Une vue en miroir en quelque sorte, qui peut être rapproché d'une solution de partage d'écran. Les deux principaux avantages relatifs à ce type d'interface, pointés par Ellis, sont sa facilité de mise œuvre et le fait qu'elle permette un fort sentiment de partage du contexte, les participants pouvant par exemple se référer à un même élément grâce à leur position spatiale [Ellis & al., 1991]. Cette approche présente néanmoins l'inconvénient d'être très peu flexible et peu adaptable au contexte de visualisation et d'édition pour les utilisateurs. Toujours selon les mêmes auteurs, l'expérience montre que les utilisateurs veulent souvent avoir un contrôle indépendant sur des paramètres de vue tels que la position ou la taille de la fenêtre, mais aussi des informations personnalisées au sein de leur interface qui serait donc différentes selon les utilisateurs. Dans les logiciels Pure Data et Max les informations telles que la taille de la fenêtre du patch ou sa position sont stockées directement au sein du document [Figure 9, p. 60]. Ainsi, l'ouverture d'un patch créé par une tierce personne amène aussi à utiliser les réglages correspondant à la vue de cette personne.

Des approches plus flexible au principe stricte de *WYSIWIS* ont alors été proposées. Selon M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, et D. Tatar, ce principe peut être relâché suivant quatre dimensions [Stefik & al., 1987] :

- Espace d'affichage : l'affichage des éléments d'interface sur lequel le principe *WYSIWIS* est appliqué.
- Temps d'affichage : à quel moment sont synchronisées les différentes vues.
- Sous-groupes : l'ensemble des participants impliqués ou touchés.
- La congruence de la vue : la congruence visuelle des informations affichées.

On parlera alors pour désigner ce genre d'interface d'un principe de WYSIAWIS ou « What You See Is Almost What I See », que l'on peut traduire par « Ce que vous voyez est *à peu près* ce que je vois ». Ce principe est adapté pour décrire des solutions logicielles qui permettent un partage de l'espace de production avec un contexte de rendu graphique qui peut être spécifique par utilisateur. Une illustration de ce principe se trouve par exemple dans l'application collaborative d'édition de texte *Google Docs* qui permet à chaque utilisateur de visualiser des parties différentes du même document ou encore d'afficher les sélections et les curseurs respectifs dans des couleurs spécifiques. Pour répondre aux différentes problématiques liées à la mise en place d'une conscience de groupe, une solution collaborative de patching audio devrait offrir des outils similaires permettant aux utilisateurs de visualiser des parties différentes d'un même patch et de différencier leurs propres sélections de celles des autres participants.

Cependant, tout ne doit pas forcément être partagé dans un logiciel multi-utilisateur. L'utilisateur peut certes disposer d'un espace commun (la représentation graphique de l'espace de production) dans lequel les données sont partagées avec les autres participants, mais il peut aussi avoir besoin d'un contexte qui lui est propre et qui ne sera donc pas partagé avec les autres. Ce contexte local peut se situer en dehors de l'espace de production (*eg.* réglages spécifiques de l'application, zone de travail locales) ou au sein même de cet espace (sélections etc.). Comme nous l'avons déjà évoqué à l'occasion de l'étude de la solution *peerdata*, la mise en place d'un contexte spécifique par utilisateur est aussi nécessaire pour le système d'*undo/redo* qui dans le cadre d'une application doit être spécifique pour chaque utilisateur [Prakash & Knister, 1992].

D’autre part, un patch n’a pas pour seule vocation d’être visualisé au sein de l’interface graphique, il fait aussi intervenir ce que l’on pourrait nommer une vue logique qui se charge de son exécution. L’application interprète le modèle d’un patch pour générer un rendu qui n’est pas seulement graphique mais aussi sonore. On pourrait donc étendre le principe de *WYSIWIS* dans le cas d’un logiciel de patching et définir ainsi d’autres notions du même ordre, telles qu’un principe de *WYHIWIH* pour « What You Hear Is What I Hear » que l’on traduirait alors littéralement par « Ce que vous entendez est ce que j’entends ». Par extension un relâchement de ce principe correspondrait alors à « Ce que vous entendez est *à peu près* ce que j’entends ». S’il est permis et même souhaité d’avoir un rendu graphique du patch qui puisse différer légèrement et s’adapter en fonction de chaque utilisateur, qu’en est-t-il du rendu sonore du patch ? Les utilisateurs cherchent-ils tout le temps à synchroniser le rendu sonore d’un patch ou veulent-ils aussi parfois pouvoir disposer de leur propre contexte d’exécution du traitement sur leur machine ? Dans des solutions telles que *peerdata* c’est l’ensemble de l’application Pure Data qui est partagé, il n’y a donc qu’un contexte d’exécution global, autrement dît et pour reprendre la formulation liée au principe de *WYSIWIS* : ce que vous voyez, faites ou entendez sur l’application a un impact direct et correspond automatiquement à ce que je vois, fait ou entends. Les messages postés dans la console, les sélections au sein du patch, les valeurs des paramètres de contrôle des différents patchs ou encore le fait que le traitement audio soit activé ou non au sein de l’application sont des éléments qui sont communs au groupe utilisant le collecticiel et ne peuvent pas être spécifiques pour chaque utilisateur. Or, lorsque l’on cherche à élaborer un traitement avec quelqu’un d’autre on peut vouloir par exemple disposer soi-même du choix d’entendre ou non son rendu sonore, ou encore vouloir disposer de ses propres valeurs de jeu pour tester le traitement mis en œuvre sans que cela ne vienne influencer l’exécution du patch chez un autre collaborateur. Si l’on considère que l’espace de jeu – celui des différents paramètres de contrôle – doit pouvoir être local au sein d’un patch dans le cadre d’un logiciel

de patching audio collaboratif synchrone, on doit alors aussi faire en sorte que l'architecture mise en place puisse supporter un contexte d'exécution spécifique qui puisse différer chez chacun des utilisateurs.

3.8. Vers une nouvelle solution collaborative de patching

Nous avons fourni dans cette première partie de mémoire un aperçu général du sujet afin de définir notre objet d'étude et d'offrir des jalons conceptuels permettant d'analyser et de classer les systèmes multi-utilisateurs. Nous avons ensuite étudié en détail une série de pratiques collaboratives utilisant les environnements logiciels actuels de patching audio. Cette étude a alors fait émergé certaines problématiques, notamment techniques, relatives à la conception des outils existants qui se proposent de les soutenir. Nous avons alors tenté de regrouper et de formaliser les différents enjeux liés à la conception d'une application collaborative de patching audio dans cette dernière partie. Aussi, quel bilan peut-on tirer de ces enseignements ? Et quel type de solution logicielle souhaitons-nous créer au regard des enjeux artistiques et pédagogiques spécifiques du projet MUSICOLL ?

Le premier constat que nous pouvons dresser à la lumière des différents outils précédemment étudiés est qu'aucune des solutions existantes n'est assez flexible pour soutenir l'ensemble des pratiques collaboratives. C'est-à-dire servir à la fois d'écrin pour l'activité d'édition et de contrôle du patch, mais aussi supporter une interaction qui pourrait se faire à la fois de manière synchrone ou asynchrone, colocalisée ou géographiquement distribuée. D'un côté, les solutions asynchrones sont limitées à l'activité d'édition du patch et ne permettent pas de se retrouver au sein d'un même espace de travail pour collaborer en temps réel. De l'autre, les solutions synchrones ne permettent pas la conception collaborative de patch en temps différé. Les utilisateurs sont obligés de se trouver au même moment sur le logiciel pour pouvoir y travailler

de concert, et aucun système de sauvegarde des documents en ligne ne permet de retrouver les patchs à des moments différents. Il nous semble aussi ressortir de cette étude que l'ensemble des solutions logicielles disponibles pour le patching collaboratif synchrone ont été créées spécifiquement à destination du jeu et de la performance collective en temps réel. Les pratiques de *live coding* avec *peerdata* [Zmölnig, 2007], [Pd~Graz, 2009], ou encore de *jam session* avec *netpd* [Haefeli, 2013] s'inscrivent par exemple entièrement dans ce type de démarche. Il existe néanmoins d'autre cas d'usage où l'aspect de conception collaborative d'un patch peut primer sur la dimension de jeu collaboratif.

Dans le cadre d'un cours d'initiation à la programmation temps réel traditionnel – tel qu'il est par exemple dispensé à l'Université Paris 8 avec des logiciels comme Pure Data ou Max comme support – on enseigne, entre autres, les processus sous-jacents aux traitements sonores, leur généralisation en quelque sorte. Cette transmission a évidemment pour but final d'amener les étudiants vers l'activité de création musicale, néanmoins elle passe essentiellement par l'activité intermédiaire de conception de l'instrument numérique. On cherche à transmettre le principe d'une modulation de fréquence, comment on construit un *flanger*, un traitement de granulation, comment ils peuvent être utilisés mais aussi comment on les fabrique, c'est-à-dire la manière dont ils peuvent être mis en œuvre, l'algorithmique qui leur est propre. L'exécution du patch ou de l'instrument créé, son rendu sonore au moment du cours, est alors placé au second plan. Il sert seulement à appuyer le propos en illustrant de manière sonore les potentialités musicales de l'instrument. Bien souvent, les étudiants ont envie de s'approprier le traitement ou le procédé mis en œuvre au sein du patch en le testant par eux-mêmes avec leurs propres fichiers sons en entrée ou en ajustant comme ils le souhaitent les différents paramètres de contrôle. Et pour cela, ils tentent de reproduire sur leurs propres machines le patch que le professeur élabore devant eux. Cependant ils abandonnent fréquemment cette tâche en raison

de la rapidité et la dextérité nécessaire à son exécution. L'enjeu d'une nouvelle solution de patching audio, qui puisse soutenir l'activité d'édition collaborative de patch dans le cadre d'un cours, était donc de rendre en ce sens les étudiants plus actifs en leur permettant d'interagir directement sur le traitement sonore et favoriser ainsi l'apprentissage par le faire-ensemble.

Lorsque nous avons imaginé la solution optimale qui viendrait soutenir cette pratique, nous nous sommes rendu compte que l'exécution du traitement devait rester locale et que les valeurs des différents paramètres de jeu ne devaient pas forcément être partagées de manière systématique et automatique entre tous les collaborateurs au sein d'un patch. C'est, dans ce cas précis, l'activité de lutherie numérique que l'on cherche à partager et non pas prioritairement l'activité de jeu sur l'instrument créé, même si elle est nécessaire dans un second temps. D'autre part, cette solution devait se situer dans les quatre quadrants de la matrice d'Ellis [Tableau 1, p. 29], ceci pour répondre notamment à des cas d'usage et d'interaction bien précis et diversifiés :

- Synchrone et colocalisé : les étudiants sont réunis en même temps et au même endroit comme en cours. Ils peuvent interagir en temps réel sur les patchs de leurs pairs ou sur celui qu'expose le professeur. Ce dernier peut aussi aider les étudiants en se connectant directement à leurs patchs ou encore exposer une difficulté rencontrée par un groupe d'étudiant sur un patch spécifique à l'ensemble de la classe en le projetant sur un écran.

- Synchrone et distribuée géographiquement : les étudiants sont réunis en même temps mais à des endroits différents pour des séances de travail à distance avec le professeur ou entre un groupe d'étudiants. Ils peuvent interagir en temps réel et à distance sur les patchs de leurs pairs ou sur celui qu'expose le professeur.

- Asynchrone et colocalisé : un étudiant absent durant une séance de cours peut poursuivre un travail entamé précédemment et s'apercevoir des modifications apportées aux documents par d'autres collaborateurs durant son absence.

- Asynchrone et distribué géographiquement : les étudiants sont à des endroits différents et à des moments différents. Ils peuvent, suite au cours en présentiel, reprendre chez eux ou ailleurs leur travail sur les patchs ou permettre au professeur d'accéder à distance à leurs travaux.

L'autre constat que nous pouvons faire au regard des différentes solutions existantes étudiées porte sur la diversité des choix techniques et des architectures mises en place par les différents projets, mais surtout de leur impact sur les fonctions qu'elles permettent d'offrir ou non aux utilisateurs. L'enjeu technique principal lié à la conception d'une nouvelle solution collaborative en temps réel que nous avons pu dégager dans ce chapitre semble être de pouvoir disposer d'une solution qui permette la réplique des données et non pas simplement un partage d'application. Ce type d'architecture apparaît en fait comme la condition initiale à la résolution des autres problématiques qui sont, par exemple, le fait de devoir fournir des éléments permettant une conscience de groupe, de réduire les problèmes de latence liés à des *temps de réponse* trop élevés et de fournir un contexte spécifique par utilisateur (à la fois au niveau du rendu graphique et sonore). La seule solution qui met en œuvre ce type d'architecture actuellement (sans forcément répondre aux autres enjeux) est *netpd*. Cette solution comporte l'avantage de pouvoir continuer à se servir de l'application Pure Data sans avoir à la modifier pour effectuer une performance musicale collective en réseau. Or cette solution n'est pas du tout conçue pour supporter l'activité de conception d'un patch de manière collaborative en

temps réel, et supporte encore moins la collaboration asynchrone. Dès lors, le fait d'envisager une nouvelle solution pour la soutenir apparaissait d'autant plus nécessaire.

Comme le note très justement M. Puckette, « nous sommes aujourd'hui tiraillés entre la volonté de dépasser les limites de ce qui se fait en matière de logiciel (ce qui implique vraisemblablement de trouver des solutions novatrices et originales aux problèmes) et le besoin de stabilité au niveau des outils (qui permettrait de préserver notre stock de documents électroniques, œuvres d'art y compris) »[87]. Le point sur lequel met ici l'accent l'auteur du logiciel Pure Data est celui de la pérennité, à la fois des logiciels eux-mêmes mais aussi des documents qui ont été construits avec. Aussi nous a-t-il fallut évaluer au début du projet si le besoin en matière d'innovation, qui se traduisait pour nous dans l'apport d'une dimension collaborative aux logiciels de patching traditionnels, était supérieur à celui de la préservation de l'existant. Cette première partie nous a permis de nous apercevoir qu'aucune solution n'était actuellement disponible pour supporter des fonctions d'édition collaborative distribuée et synchrone de patch. Deux possibilités s'offraient alors à nous en début de projet : partir d'une solution de patching existante puis l'adapter, ou bien développer notre propre solution en repartant de zéro.

La possibilité de partir du logiciel Max pour y *apporter* des modalités collaboratives à tout de suite été exclue du fait de son caractère propriétaire qui nous aurait empêché d'accéder à ses fondements pour y effectuer les modifications nécessaires. D'autre part, ce logiciel n'est pas multiplateforme[88], ce qui représentait une des spécifications initiales à la solution que nous

[87] [Puckette, 2009], p. 195.

[88] Le logiciel Max est disponible sur les plateforme *MacOS* et *Windows*, mais par exemple pas sur *Linux*.

voulions développer dans le cadre du projet MUSICOLL. L'approche consistant à partir du logiciel Pure Data pour y effectuer des modifications a en revanche été envisagée car elle présentait de réels intérêts. Cette approche aurait alors consisté à effectuer un *fork*[89] du logiciel libre Pure Data puis d'effectuer des modifications à son code source. Elle nous aurait permis de bénéficier d'une communauté d'utilisateurs déjà présente et conséquente réunie autour du logiciel ; de maintenir une rétrocompatibilité des documents existants ; de bénéficier des fonctionnalités essentielles déjà mises en œuvre ; et, dans une démarche allant dans le sens de la création de logiciel *open-source*, de pouvoir à la fois contribuer au code source original et de bénéficier en retour des apports des contributeurs actifs du projet, ce qui aurait ainsi pu améliorer la pérennité des différents développements. Elle n'a néanmoins pas été retenue, ceci pour plusieurs raisons. Le développement d'une solution de patching collaborative telle que nous l'envisagions aurait impliqué de faire de nombreux changements, notamment au niveau du noyau fonctionnel du logiciel, de son interface graphique. D'autre part nous souhaitions aussi revoir certains éléments du langage pour l'adapter à un contexte pédagogique. Ces choix auraient alors dû être discutés avec la communauté de développeurs existante sur le logiciel, qui ne les auraient alors pas forcément validés, freinant d'autant le développement et nous amenant alors à devoir maintenir une version détachée de la solution Pure Data qui aurait alors été dans le sens inverse des intérêts en matière de pérennité exposés plus haut[90]. Maintenir une rétrocompatibilité avec les documents préexistants aurait aussi représenté une réelle contrainte de développement qui nous aurait empêché d'apporter tout changements radicaux au niveau du

[89] Un *fork*, dans le domaine du développement logiciel provient d'une scission d'un projet initial, partageant avec lui une partie de son code source.

[90] *Pd-Extended* (https://puredata.info/downloads/pd-extended, consulté le 04/09/18), qui a longtemps été un *fork* très utilisé du logiciel Pure Data est aujourd'hui un projet considéré comme obsolète qui n'est plus maintenu depuis plus de cinq ans. Il représente en ce sens une illustration de ce qui nous paraît être un écueil notoire lié à ce type d'approche.

langage au sein du patch ou du logiciel de manière générale. D'autre part, l'analyse du noyau de Pure Data et de son interface graphique effectuée par l'équipe, notamment à l'occasion du projet de développement HOA [Guillot, 2014], nous avait amené au début du projet à la conclusion que beaucoup de choses demanderaient à être revues au niveau de son code source si nous l'utilisions[91].

Pour l'ensemble de ces raisons, mais aussi dans une démarche d'autoformation par la recherche et le développement, le choix a donc été fait de créer une nouvelle solution en partant de zéro, pour faire face aux différents défis techniques rencontrés dans cette première partie de mémoire, et répondre de manière prioritaire aux enjeux en matière de pédagogie, à savoir ceux de l'élaboration collaborative de traitements sonores. Dans la prochaine partie de ce document, nous présenterons donc cette nouvelle solution logicielle et donnerons les détails de sa mise en œuvre.

[91] Les éléments bloquants étaient, de manière non-exhaustive, le fait qu'il n'y avait pas de support pour le multi-instance (il était alors prévu au début du projet que l'application puisse être intégré sous la forme d'un plugiciel à des stations de travail audionumériques), que les opérations d'*undo/redo* étaient limitées à une seule action, que le code était en langage C et que celui-ci était peu documenté ou encore le fait que l'interface graphique soit mise en œuvre avec le langage de script *Tcl/Tk* (https://www.tcl.tk/ [consulté le 05/09/18]) aujourd'hui de moins en moins adapté à la création d'interfaces graphiques modernes. Notons néanmoins que ce constat a été établi au début du projet il y a de ça plus de quatre ans, et que beaucoup de choses ont depuis été améliorées ou sont en cours d'amélioration au sein de Pure Data ou par différents projets connexes. Le support du multi-instance et de l'*undo/redo* illimité est actuellement en cours d'intégration au sein de Pure Data. Des *forks* de Pure Data tels que *Spaghettis* (https://github.com/Spaghettis/Spaghettis [consulté le 05/09/18]) visent à l'amélioration de la documentation du code, enfin d'autres comme *Purr Data* (https://github.com/agraef/purr-data [consulté le 05/09/18]) visent notamment la réécriture de son interface graphique.

Partie II – Présentation et mise en œuvre de l'application Kiwi

Kiwi est une application de bureau, disponible pour les plateformes Mac, Windows et Linux[92]. Elle permet l'édition et l'exécution de patch de synthèse ou de traitements sonores en temps réel et se situe donc à ce titre dans la lignée des environnements de patching traditionnels tels que Max ou Pure Data. Si une utilisation locale de l'application est permise, son originalité se situe surtout dans le fait qu'elle supporte aussi une utilisation en ligne, permettant alors une collaboration synchrone et à distance entre plusieurs utilisateurs sur un même patch. Dans cette configuration, chaque application cliente peut être connectée à un ou plusieurs patchs, que nous appelons aussi document, via à un serveur central qui permet d'orchestrer le travail collaboratif entre les différents utilisateurs du collecticiel.

Le développement de Kiwi a dû faire face à deux enjeux généraux. Le premier était de concevoir un nouveau logiciel de patching audio qui soit fonctionnel et qui puisse offrir aux utilisateurs les opérations de base attendues au sein d'un environnement de ce type. Le but étant, dans le cadre du projet MUSICOLL, d'obtenir une première maquette viable qui permettrait notamment d'effectuer une transition depuis les environnements traditionnels vers ce nouveau support logiciel dans le cadre d'un cours pilote d'initiation au patching audio repensé par l'équipe, puis donné durant le second semestre 2018 à l'Université Paris 8 [Galleron & al., 2018]. Ce premier enjeu était de taille dans la mesure où beaucoup de choses demandaient à être créées : de l'interface graphique générale à son moteur logique et sonore en passant par la définition du langage graphique du patch. Si une grande partie du développement a été consacrée à ces enjeux techniques durant le projet, le cadre de ce mémoire de thèse, centré

[92] L'application Kiwi est téléchargeable à l'adresse suivante : https://github.com/Musicoll/Kiwi/releases.

essentiellement sur les modalités collaboratives apportées par la solution, ne nous permettra pas de rentrer en profondeur dans les détails de leurs mises en œuvre. Aussi, certains de ces aspects ne seront traités que du point de vue de l'utilisateur omettant celui du concepteur, pour nous permettre de concentrer notre exposé sur les aspects fonctionnels et techniques plus spécifiques aux seconds enjeux de développement, à savoir ceux liés à la dimension collaborative.

L'application Kiwi a été développée en langage C++, à l'aide de la bibliothèque *Juce*[93] – une infrastructure applicative multiplateforme adaptée à la création d'interface graphique et au traitement du signal – mais aussi avec la bibliothèque *flip*, une infrastructure logicielle mise à notre disposition par le partenaire industriel *OhmForce* dans le cadre du projet MUSICOLL[94]. Cette dernière a représenté pour nous une brique essentielle à la création de l'application, nous permettant de répondre à de nombreux enjeux évoqués dans la première partie de ce mémoire. Elle permet notamment la modélisation d'un document, le patch, qui peut être utilisé localement ou à plusieurs en temps réel. Elle dispose d'un mécanisme de synchronisation et de gestion de l'édition concurrente des documents sur plusieurs sites distants, allant dans le sens des enjeux évoqués en première partie et donc de l'architecture que nous voulions adopter dans le cadre de la création de Kiwi. Nous aurons l'occasion de préciser son fonctionnement général et la manière dont nous nous en sommes servis au cours de la présentation de cette application.

---

[93] La bibliothèque *Juce* possède une licence permissive qui autorise une utilisation gratuite dans un contexte *open-source*. https://juce.com/ [Consulté le 05/09/18].

[94] Pour permettre au code source de l'application Kiwi d'être librement exposé sur internet, celle-ci y a été liée et intégrée sous la forme d'une bibliothèque dynamique.

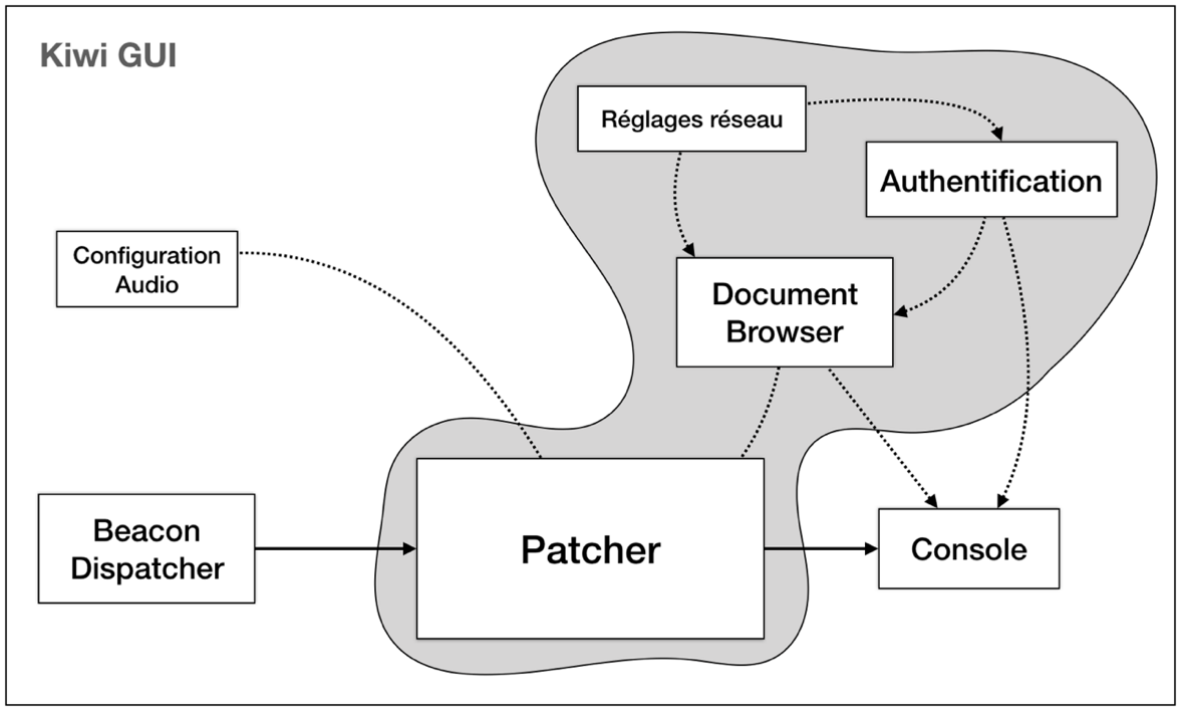

Figure 17 - Schéma offrant une vue d'ensemble des différents composants de l'interface graphique utilisateur (GUI) de l'application cliente de Kiwi et leur interrelations. Les liens symbolisent les interactions fortes (en gras) ou faibles (en pointillés) qui existent entre elles. La zone grisée comprend les composants spécifiques à la mise en place des aspects collaboratifs de l'application, tandis que ceux à l'extérieur correspondent plus à ce que l'on trouve dans les logiciels traditionnels de patching (exception faite de la fenêtre *Patcher* qui se trouve être dans les deux catégories à la fois).

Les trois chapitres qui structurent le nœud de ce mémoire de thèse suivent la trame offerte par l'outil conceptuel du trèfle des systèmes multi-utilisateurs que nous avons exposé dans la première partie de ce mémoire [Ellis & Wainer, 1994], [Salber, 1995], et illustré à la [Figure 3, p. 31]. Cette présentation générale sera aussi guidée par l'interface graphique du logiciel et des différents composants qui la composent [Figure 17].

Dans un premier temps, nous aborderons les questions relatives à la conception de l'espace de production, c'est-à-dire de l'espace, graphique et logiciel, qui permet à l'utilisateur d'effectuer les opérations essentielles à la conception d'un patch et à son exécution. Nous discuterons des choix méthodologiques et généraux liés à l'élaboration du langage graphique de Kiwi, de

l'interface graphique générale du logiciel (fenêtre du *Patcher*, de la *Console*, du *Beacon Dispatcher* et de *Configuration Audio* [Figure 17]) et des opérations qu'il offre dans un contexte à la fois mono et multi-utilisateur. Nous y aborderons aussi des points plus techniques, relatifs notamment à la modélisation du document et à l'architecture logicielle de l'application, dont les enjeux principaux étaient de permettre le partage des données d'édition du patch entre les utilisateurs, tout en leur permettant de garder un contexte d'exécution local.

Dans un second temps nous nous concentrerons sur les aspects de conception de l'espace de coordination de l'environnement Kiwi. Cet espace est celui qui permet d'orchestrer le travail collaboratif et d'aider les utilisateurs à réaliser des actions de groupe. Nous aurons alors l'occasion d'évoquer plus en détail l'architecture réseau de Kiwi, les aspects relatifs au serveur et à la gestion des sessions et d'exposer les composants de l'interface graphique qui permettent cette coordination (fenêtre du *Patcher*, de *Réglages réseau*, d'*Authentification* et *Document Browser* [Figure 17]). Nous étudierons aussi les solutions qu'apporte Kiwi en matière de gestion de la concurrence et les mécanismes de conscience de groupe qui ont été mis en place principalement au sein du composant graphique du patch afin de permettre aux utilisateurs de coordonner cette activité de conception.

Le dernier chapitre de cette partie est dédié à l'espace de communication. Celui-ci est beaucoup moins conséquent que les deux précédents. Cela s'explique par le fait que nous nous sommes prioritairement consacrés au sein du projet sur les dimensions de conception du patch, sur son édition, et surtout sur la manière de rendre viable cette pratique collaborative, en écartant dans un premier temps la dimension du jeu collaboratif. Au cours du projet, nous avons néanmoins décidé d'y apporter une première réponse, qui s'est soldée notamment par l'ajout d'un nouvel objet, que nous avons nommé *hub*, permettant une communication inter-utilisateur à travers un

document. Nous aborderons alors les détails de sa mise en œuvre et les possibilités offertes par cet objet.

La conception de Kiwi a été un processus itératif, alimenté à la fois par l'évolution du code de l'application et de l'écosystème du logiciel en général, mais aussi par les retours des premiers utilisateurs ; à savoir les enseignants et chercheurs du CICM, les étudiants de l'Université Paris 8 et les différents compositeurs ayant utilisé l'application. Le logiciel, notamment son interface graphique mais aussi son architecture client et client-serveur, a subi plusieurs réusinages[95] au cours du processus de développement. Ces modifications étaient nécessaires pour introduire certaines nouvelles fonctionnalités mais aussi pour corriger des erreurs liées à l'architecture même de l'application. Il sera donc aussi intéressant de revenir sur les principaux changements opérés au cours des différentes versions de l'application.

4. Conception de l'espace de production

L'espace de production d'un logiciel de patching est constitué de plusieurs composants graphiques qui ont pour but d'assister l'utilisateur dans la tâche de conception et de contrôle des différents patchs de l'environnement. L'application Kiwi, telle qu'elle se présente notamment dans le cadre d'une utilisation mono-utilisateur, entretient beaucoup de similitudes avec les logiciels Max, Pure Data ou encore *Purr Data*. De manière générale, nous n'avons pas cherché à nous démarquer fondamentalement de ces solutions existantes. Cela était important pour nous, afin de permettre à des utilisateurs expérimentés de garder leurs points de repères et de prendre ainsi plus facilement en main ce nouveau logiciel. Mais aussi pour permettre aux

[95] Nous employons ce terme pour désigner ce que l'on nomme plus couramment le *refactoring* dans le domaine du développement logiciel.

utilisateurs débutant de pouvoir plus aisément opérer une transposition de leur pratique acquise au sein de Kiwi vers les environnements traditionnels s'ils le souhaitaient. Des différences sont néanmoins notables, notamment au niveau du langage fonctionnel et graphique comme nous le verrons dans la première section (4.1), ou au niveau de la mise en œuvre des composants graphiques secondaires de l'interface (4.4). L'espace de production, tel que le définissent plus spécifiquement C. A. Ellis et J. Wainer [Ellis & Wainer, 1994], [Figure 3, p. 31], désigne les objets accessibles, produits ou manipulés par l'utilisateur d'un logiciel, ou par le groupe au sein d'un collecticiel, ainsi que les opérations possibles sur ces objets. Il se concentre alors dans notre cas sur le composant central du logiciel à savoir le patch, en tant que document visualisable, modifiable et utilisable. L'activité de conception d'un patch dans Kiwi, qu'elle soit partagée ou locale, passe par la modification de ce document. La manière dont il a été modélisé, à l'aide de la bibliothèque *flip*, et l'architecture logicielle que nous avons mise en place, nous ont permis de définir ce qui devait être partagé entre les utilisateurs de ce qui resterait local dans le contexte d'une session collaborative. La section (4.2) sera donc consacrée à ces aspects techniques de conception. Ils sont fondamentaux dans la mesure où ils déterminent la manière dont se comporte l'application et les fonctionnalités qu'elle permet d'offrir aux utilisateurs dans le contexte de l'espace commun du patch. Ceci nous permettra d'aborder alors la section suivante (4.3) qui sera quant à elle centrée sur les opérations offertes au sein de l'espace de production partagé et local du point de vue plus spécifique de l'utilisateur.

4.1. Choix généraux du langage Patcher de Kiwi

La pratique du patching requière un apprentissage à long terme de la part des utilisateurs, mais c'est aussi le rôle du concepteur et du développeur du logiciel que de prendre en compte les potentielles difficultés afin de faciliter au maximum la compréhension et l'utilisation de ce langage de programmation graphique. Deux enjeux principaux ont donc guidé le

développement de l'interface graphique générale de l'application, et du Patcher [96] en particulier : apporter une bonne lisibilité au traitement mis en œuvre, et assurer une bonne utilisabilité de l'outil. La lisibilité d'un traitement au sein d'un patch passe par la manière dont sont représentés les objets, les liens, les interactions ou encore les valeurs de jeu, mais aussi par la possibilité pour l'utilisateur d'arranger comme il le souhaite les éléments graphiques qui le composent pour assurer une meilleure intelligibilité du processus mis en œuvre. L'utilisabilité, quant à elle, peut se définir dans notre cas comme le degré de complexité à réaliser une action spécifique au sein d'un patch ou de l'application en général. L'enjeu de développement n'était pas de recréer l'ensemble des fonctionnalités du langage Patcher, tel qu'il est mis en œuvre dans les solutions historiques de patching, mais plutôt d'offrir un outil maniable avec suffisamment de fonctionnalités pour satisfaire les besoins spécifiques du projet. L'expression de ces besoins s'est faite par rapport à notre propre pratique, mais aussi à travers l'analyse des cours d'introduction aux langages de patching donnés par les différents membres du CICM à l'Université Paris 8. Cela nous a permis de bénéficier d'une expérience d'équipe importante au sein du projet qui nous a amené à repenser certains éléments du langage pour les adapter à un contexte pédagogique, à effectuer certaines spécifications du point de vue du développement et à déterminer quelles fonctions seraient nécessaires au sein du logiciel pour pouvoir opérer une transition du support logiciel utilisé initialement pour les cours vers la nouvelle solution Kiwi. Nous détaillons dans les trois prochaines sous-sections l'approche méthodologique qui a dicté la création des objets de Kiwi (4.1.1), mais aussi les choix graphiques (4.1.2) et fonctionnels (4.1.3) des éléments composant son langage.

[96] Nous désignons sous le terme Patcher le composant graphique permettant de visualiser et d'interagir avec un patch au sein de l'environnement logiciel.

4.1.1. Objets disponibles

Les objets instanciables par les utilisateurs au sein d'un patch Kiwi ont été ajoutés progressivement à l'application. Le choix des objets que nous devions ajouter mais aussi dans quel ordre ils devaient l'être a été dicté par plusieurs impératifs ; en premier lieu pour guider le développement exploratoire de l'application et ensuite pour répondre aux enjeux de la création musicale et de la pédagogie liée à la pratique du traitement audio en temps réel.

Dans les premières versions de l'application, l'interface graphique de l'application était plutôt sommaire. Elle ne comportait par exemple pas d'objet de contrôle permettant d'envoyer des messages à partir du patch, aussi, la première version de l'application se composait uniquement de cinq objets : l'objet *newbox* (qui permet la création de nouveaux objets via un éditeur de texte intégré), l'objet *errorbox* (qui permet de notifier qu'un objet est non-valide au sein du langage graphique), l'objet *receive* (qui permet de recevoir des messages depuis l'extérieur du patch), l'objet *print* (qui permet d'envoyer les messages reçus au sein du patch vers l'extérieur, par exemple dans l'interface de console[97]) enfin l'objet *plus* (qui permet d'additionner deux nombres). Ces cinq premiers objets nous ont alors permis de développer la logique de transmission des messages au sein du patch ainsi que l'interface graphique globale du logiciel. Dans un second temps, lors de la mise en place du traitement du signal au sein du patch, nous avons ajouté les objets *adc~* et *dac~*[98] (qui permettent respectivement de récupérer et d'envoyer des échantillons sonores entre le patch et la carte son de la machine) puis quelques objets mathématiques tels que le *times~* (qui permet de multiplier deux vecteurs d'échantillons). Des objets tels que le *metro* (qui permet de sortir un message *bang* à intervalles réguliers) ou encore

[97] Ce composant sera abordé en détail dans la sous-section 4.4.1 (p. 153) de ce document.

[98] Acronymes de langue anglaise pour : *Analogic to Digital audio Converter* et *Digital to Analogic audio Converter*.

l'objet *pipe* (qui retarde la sortie d'un message dans le temps) ont ensuite été ajoutés lorsque nous avons dû mettre en place la gestion temporelle et le séquencement des événements au sein du patch (*scheduler*). Les objets tels que le *toggle* (interrupteur faisant sortir une valeur booléenne) ou encore le *slider* (potentiomètre linéaire) ont été créés et ajoutés lors de la phase de développement relative aux objets graphiques. L'objet *meter~* a aussi été ajouté à ce moment là pour nous permettre d'obtenir un rendu graphique du signal et car il représentait un défi technique à relever, faisant intervenir à la fois des dimensions de traitement du signal et d'interface graphique.

Lorsque ces fonctionnalités initiales essentielles ont été présentes au sein de l'application, nous avons alors pu envisager la création de nouveaux objets pour rendre le patch utilisable dans un contexte de création musicale et de pédagogie en augmentant le nombre d'objets disponibles. Pour cela, l'équipe a énuméré les principaux concepts généraux enseignés lors des précédentes séances de cours donnés avec les environnements traditionnels de patching à l'Université. Nous en avons sorti une liste de fonctions essentielles que le logiciel devrait avoir ainsi qu'une liste d'objets, classés en fonction de leur intérêt pédagogique ou de leur utilité : *indispensable*, *souhaitable*, *non-nécessaire*. Les priorités de développement au niveau des objets ont alors été établies en fonction. Nous listons ici quelques-uns de ces concepts généraux de manière non-exhaustive et des exemples d'objets qui ont été créés afin de soutenir leur transmission dans le cadre du cours de Kiwi :

- Principes généraux liés au traitement du signal : nécessitait une suite d'opérateurs arithmétiques et logiques (*e.g.* +~, -~, /~, *~, ==~, >~), mais aussi l'objet *sig~* pour générer un signal ou encore l'objet *snapshot~*, pour permettre de visualiser la valeur d'un signal.

- Organisation, contrôle du patch et transmission de *bonnes pratiques* : lissage du signal avec l'objet *line~*, mais aussi possibilité d'apposer des commentaires grâce à l'objet *comment* pour clarifier son patch, de séquencer proprement les messages à l'aide de l'objet *trigger* par exemple ou encore de gérer les variables d'initialisation au chargement du patch avec l'objet *loadmess*.
- Gestion des entrées et des sorties du logiciel : avec les objets *adc~* et *dac~* précédemment cités, *sf.play~* et *sf.record~*[99] qui permettent de lire des fichiers sonores à partir du disque de la machine et d'enregistrer des fichiers sonores sur le disque.
- Synthèse additive, synthèse par modulation d'amplitude ou de fréquence : objets *osc~* (oscilateur sinusoïdal), objet *phasor~* (générateur de rampe) etc…
- Génération d'enveloppes d'amplitude : objet *line~* mais aussi l'objet *message* qui permet de formater des listes d'élément pour les envoyer aux autres objets.
- Création de traitements sonores fondés sur la ligne à retard et la réinjection de signal comme le *flanger*, le *chorus* ou encore des filtres en peigne : pour cela l'objet *delaysimple~* a été créé (objet de haut niveau intégrant à la fois des fonctionnalités de retard et de réinjection).
- Gestion de l'aléatoire : objet *random* (génération de nombres pseudo-aléatoires).
- Modalités collaboratives propres à Kiwi, notamment via l'objet *hub*[100].

[99] Notons que ces deux objets spécifiques ont fait l'objet d'une forte demande de la part des étudiants et du professeur durant le cours mais n'ont été ajoutés à l'application Kiwi qu'à la version *v1.0.2* et n'ont donc pas pu être exploités durant le semestre.

[100] Nous présenterons cet objet dans la section 6.1 (p. 215) de ce document.

L'application Kiwi comprend actuellement environ 70 objets instanciables au sein d'un patch[101]. Des patchs d'aide explicitant le fonctionnement de chacun de ces objets ont été créés, néanmoins, contrairement à Max ou Pure Data, ils ne sont pas encore accessibles directement au sein de l'application[102]. Ils sont téléchargeables à partir d'un dossier de documentation séparé qui comprend aussi des patchs d'exemples et quelques tutoriels permettant de se familiariser avec ce nouvel environnement[103].

4.1.2. Éléments graphiques

La lisibilité d'un patch passe selon nous en grande partie par la différenciation graphique exercée entre les types d'objets pouvant y être instanciés. On distingue plusieurs types d'objets au sein d'un patch. Les objets amenés à gérer uniquement des données de type message (*e.g.* « + » ou encore *trigger*), les objets amenés à traiter du signal (*e.g.* « +~ »), enfin les objets pouvant traiter les deux types de données. La distinction entre un objet gérant des messages et un objet amené à manipuler des signaux reprend le principe graphique présent dans les logiciels Max et Pure Data à savoir, tous les objets de type signal comportent le suffixe « ~ ».

---

[101] La liste complète des objets est disponible à l'adresse suivante : http://musicoll.github.io/Kiwi/#objects-fr [consulté le 06/09/18].

[102] Ce point-là devrait faire l'objet d'un développement futur qui n'a pas encore été effectué.

[103] Les ressources liées à la documentation de l'application sont téléchargeables à l'adresse suivante : https://github.com/Musicoll/Kiwi/releases/tag/v1.0.2 [consulté le 07/09/18].

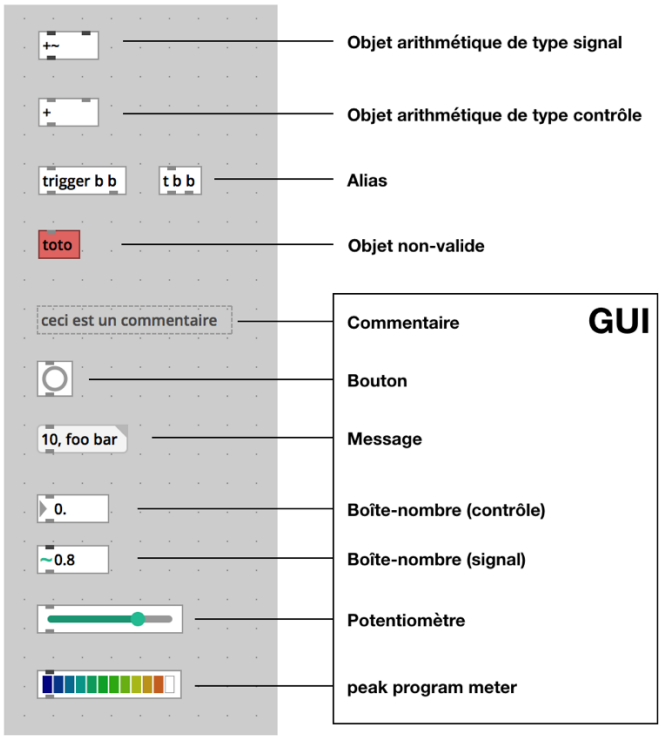

Figure 18 – Différents types d'objets pouvant être contenus dans un patch (de haut en bas : +, +~, *trigger*, *errorbox*, *comment*, *bang*, *message*, *numbox*, *numbox~*, *slider*, *meter~*). Le rectangle nommé GUI regroupe un type particulier d'objets que l'on appelle communément objets graphiques ou aussi objets de contrôle.

Tous les objets contenus dans un patch sont graphiques, néanmoins une distinction peut aussi être faite entre un objet statique, comportant uniquement du texte et avec lequel il n'est pas possible d'interagir autrement qu'avec des messages, et un objet avec lequel l'utilisateur peut interagir directement à l'aide des interfaces de contrôle de la machine telles que la souris (un potentiomètre, un bouton ou un message), du clavier (boîte-nombre), ou dont l'apparence change en fonction de son état interne (par exemple un objet *meter~* ou *number~*) [Figure 18].

Une boîte de commentaire doit pouvoir être clairement identifiée comme une entité purement informative du patch, qui n'influe pas sur son exécution. Une boîte-message au contraire doit être perçue comme un élément cliquable avec lequel on peut interagir et doit donc en ce sens

être clairement distinguée par rapport à un objet statique standard[104] ou encore à la version signal de ce même objet qui est destinée à la visualisation des valeurs d'un signal. Nous avons donc essayé, dans la mesure du possible, d'apporter un contraste au niveau de l'interface graphique entre chaque objet [Figure 18]. Le langage Patcher tel qu'il est mis en œuvre dans Kiwi supporte aussi ce que l'on appelle les alias d'objet. Les alias, disponibles aussi dans les mises en œuvre traditionnelles des environnements de patching tels que Max ou Pure Data, sont des noms alternatifs donnés aux objets pour que l'utilisateur puisse les instancier plus rapidement au sein du patch. Les alias n'ont cependant pas été généralisés à tous les objets et ne sont disponibles que dans certains cas, lorsque les noms sont trop long à écrire, ou les objets trop souvent employés dans la pratique (*e.g.* on pourra taper simplement *t* pour instancier un objet *trigger* [Figure 18], ou encore *r* pour un objet *receive*). Ces derniers sont représentatifs de l'équilibre qui existe dans la conception de l'interface graphique entre l'utilisabilité et la lisibilité du patch. Un utilisateur confirmé préfèrera peut-être la version raccourcie du nom de l'objet dans la mesure où il surcharge moins l'interface du patch, tandis qu'un utilisateur débutant préfèrera sans doute la version complète du nom, qui peut être plus explicite.

Un autre paramètre sur lequel il est possible d'agir pour donner plus de lisibilité au traitement est de différencier les types de données compatibles avec chaque entrée/sortie d'un objet. Pure Data ne fait pas la distinction dans son interface graphique entre une entrée/sortie acceptant du signal et une entrée/sortie destinée à transmettre ou recevoir uniquement des messages. D'autres mises en œuvre, comme Max ou *Purr Data* au contraire la font. Nous avons choisi de suivre ces dernières en ajoutant un léger contraste entre les types d'entrées/sorties des objets pour

[104] Il n'est pas rare que des utilisateurs débutants confondent par exemple une boîte-message avec un objet standard, et ne comprennent pas pourquoi ils n'obtiennent pas le résultat escompté lorsqu'ils cliquent sur un objet "+" pour faire sortir une valeur.

augmenter la lisibilité du patch et faciliter son utilisation [Figure 18]. Ce constat a été réalisé pendant la classe pilote de Kiwi où nous nous sommes aperçus que les étudiants avaient du mal à identifier s'ils pouvaient ou non connecter la sortie d'un objet à l'entrée d'un autre objet.

4.1.3. Spécificités fonctionnelles

Le langage Patcher tel qu'il est mis en œuvre dans Kiwi suit plus ou moins les mêmes spécifications que celui mis en œuvre au sein de Max et Pure Data, néanmoins, des différences existent au sein même de ces deux derniers environnements, même s'ils proviennent d'un paradigme commun (nom des objets, spécificités de format et de sémantique des messages). Aussi est-il nécessaire d'apporter quelques clarifications sur le langage Patcher tel qu'il est mis en œuvre dans Kiwi. Celui-ci reprend le concept d'*atom*, un type de données variant pouvant être soit un nombre flottant, soit un nombre entier, soit une chaîne de caractères. Il est de plus possible d'associer plusieurs *atoms* afin de composer une liste. Certains éléments du langage ont aussi une signification particulière calquée sur ses homologues, comme le signe *dollar* (« $ ») qui permet de formater des listes en intégrant des variables d'entrée à un message ou encore la virgule (« , ») qui permet de séparer une liste d'éléments et de les envoyer de manière séquentielle au sein d'un message. Enfin, d'autres éléments ne sont, pour l'instant, pas pris en compte dans Kiwi, comme le point-virgule « ; » qui permet dans Max et Pure Data d'envoyer un message à un objet sans avoir recours à une connexion physique.

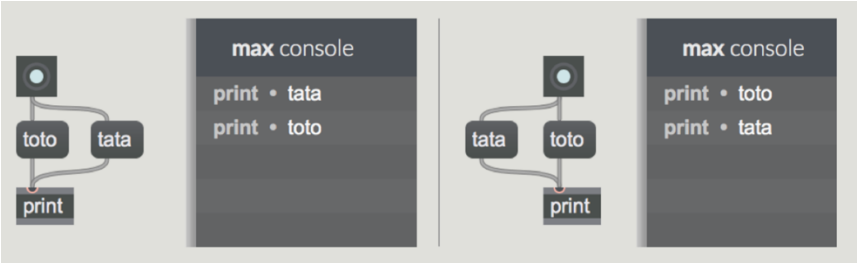

Figure 19 – Capture d'écran d'une partie de patch Max qui expose le comportement de l'ordonnancement des messages dans ce logiciel. L'ordre d'arrivée des messages à l'objet *print* dépend du placement graphique des objets.

L'ordonnancement des messages envoyés par la sortie d'un objet est indéterminé dans les spécifications données au langage Patcher de Kiwi. Dans le logiciel Max au contraire, le placement graphique des objets au sein du patch définit la manière dont sont ordonnés les messages en sortie des objets [Figure 19]. Nous aurions pu suivre le même principe dans Kiwi, néanmoins, comme le note Puckette, cette solution a le désavantage de modifier le comportement du patch, dès lors que l'on bouge des objets, ne serait-ce que pour nettoyer le patch [Puckette, 2002, p. 11]. Nous avons donc rejoint l'auteur de Pure Data sur ce point et avons préféré encourager l'emploi de l'objet *trigger* à la place, qui constitue selon nous une *bonne pratique* à enseigner aux débutants dans ce genre d'environnement pour expliciter l'ordre dans lequel sont distribués en sortie les messages envoyés par les différents objets.

Une autre spécificité du langage Patcher de Kiwi, que nous souhaitons exposer dans ce cadre, est relative à l'observabilité de l'interface du patch. S. Salber, J. Coutaz, D. Decouchant et M. Riveill définissent le principe d'observabilité comme « la capacité du système à rendre perceptibles à l'utilisateur les variables d'état internes pertinentes pour la tâche en cours » [Salber & al., 1995, p. 4]. Dans les logiciels Max ou Pure Data, mais aussi dans Kiwi, les arguments tapés à la suite du nom de l'objet au sein de la boîte servent à initialiser une valeur interne au sein d'un objet, cette valeur est le plus souvent à comprendre comme une valeur *par défaut*, c'est-à-dire une valeur servant à l'initialisation au chargement de l'objet.

Le plus souvent, les objets disposent de méthodes qui permettent de modifier cette valeur au cours de l'exécution du patch, la valeur d'initialisation est alors écrasée par la nouvelle valeur.

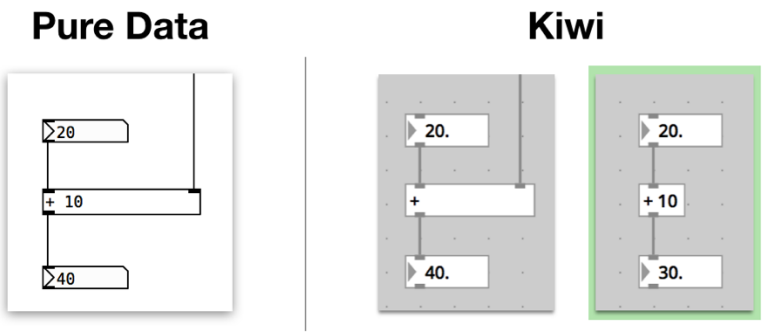

Figure 20 – Capture d'écran d'une partie de patch Pure Data (à gauche) et Kiwi (à droite) explicitant un problème d'observabilité du patch analysé à la fois dans Max et Pure Data, que nous avons cherché à résoudre dans Kiwi.

La figure ci-dessus présente une partie de patch dans le logiciel Pure Data [Figure 20, vignette de gauche] et Kiwi [Figure 20, vignette centrale], volontairement tronquée, qui expose un objet central *plus* permettant d'additionner deux nombres en entrée et d'obtenir le résultat de l'opération en sortie dans une boîte-nombre. Dans le patch Pure Data un argument a été spécifié à la suite du nom, à la création de l'objet. Il correspond en l'occurrence ici à la valeur de l'opérande de l'opération d'addition. Le comportement dans Pure Data, qui est aussi le même dans Max, est d'écraser la valeur d'initialisation par défaut de l'objet lorsqu'un message arrive dans l'entrée de droite de l'objet. Le problème est que, pour des utilisateurs novices, ce comportement peut produire des erreurs de compréhension et de lecture du flux d'exécution du patch dans la mesure où il ne permet pas d'observer la variable d'état interne de l'objet. Selon l'auteur de la documentation du logiciel Pure Data : « parmi les principes de conception de Pd [Pure Data], il y a le fait que les patchs doivent être imprimables, c'est-à-dire que l'apparence

d'un patch doit déterminer pleinement ses fonctionnalités »[105]. Dans le cas présent, le patch donne selon nous la fausse information que la valeur « 10 » va être ajoutée à un nombre et ne suit donc pas ce principe. L'auteur continue en disant qu'« il est probablement mauvais de spécifier des arguments de création tels que "*makenote 64 250*" si vous comptez les remplacer plus tard ; c'est déroutant pour quiconque essaie de comprendre le patch »[106]. Aussi dans le cadre de la création de Kiwi, et dans un enjeu de pédagogie, l'équipe a décidé de prendre une liberté avec le langage Patcher en revoyant la façon dont sont gérés et représentés les arguments des objets au sein du patch pour mieux refléter le comportement interne du patch : « le choix a été fait d'interpréter un argument comme une constante de l'objet si celui-ci est spécifié à sa création » [Paris & al., 2017, p. 5.]. La conséquence est que le nombre d'entrées des objets suivant ce principe est alors conditionnée par le fait que l'objet dispose ou non d'un argument. Par exemple, dans la figure présentée ci-dessus, l'opérande de l'objet *plus* est fixe, de cette façon, il ne peut plus être modifié durant l'exécution du patch, améliorant alors, selon nous, la lisibilité de l'opération en cours dans la mesure où l'utilisateur peut désormais se fier à ce que le patch fait apparaître graphiquement ou non [Figure 20, vignette de droite][107].

4.2. Modélisation et architecture logicielle

Dans la première partie de ce mémoire, nous avons vu que la réplique des données entre les utilisateurs représentait l'enjeu technique principal lié à la conception d'une solution

---

[105] Tiré de la documentation de Pure Data, (http://msp.ucsd.edu/Pd_documentation/x2.htm, §2.6.2, [consulté le 08/09/18]) et traduit de l'anglais : « Among the design principles of Pd is that patches should be printable, in the sense that the appearance of a patch should fully determine its functionality ».

[106] Tiré de la documentation de Pure Data, (http://msp.ucsd.edu/Pd_documentation/x2.htm, §2.6.2, [consulté le 08/09/18]) et traduit de l'anglais : « It is probably bad style to specify creation arguments ala "*makenote 64 250*" if you are going to override them later; this is confusing to anyone who tries to understand the patch ».

[107] Ce comportement a été mis en œuvre dans beaucoup d'objets de Kiwi, notamment dans tous les opérateurs arithmétiques (de type signal ou de type message), ou encore dans l'objet *sig~*. Il souffre néanmoins de quelques incohérences dans la mise en œuvre actuelle dans la mesure où il n'a pas été adapté à tous les objets.

collaborative synchrone dans la mesure où il constituait la condition initiale à la résolution des autres problématiques liée à la mise en place de la conscience de groupe ou encore à des temps de réponse trop élevés. L'autre enjeu du développement de Kiwi était de créer une solution de patching qui fournisse un contexte d'exécution pouvant être spécifique pour chaque utilisateur. La logique de l'application ne devait donc pas être centralisée comme dans les solutions que nous avons évoquées dans la première partie de ce document[108] mais bien localisée au sein de chaque application cliente. Ces différents enjeux posent des questions architecturales qui ont été résolues en grande partie grâce à l'emploi de la technologie *flip* au sein du projet. Comme nous le verrons dans cette section, cette bibliothèque nous a permis de modéliser le patch (4.2.1), de concevoir une architecture locale autour de ce modèle de données (4.2.2), et d'orchestrer la réplique des données partagées entre plusieurs sites distribués en maintenant un contrôle logique local (4.2.3).

4.2.1. Modélisation du patch

La bibliothèque *flip* permet la modélisation d'un document à partir du langage C++, par l'intermédiaire de types de données spécifiquement conçus pour supporter un accès concurrentiel[109]. La tâche du développeur utilisant cette bibliothèque est alors simplement de modéliser le document en fonction de ses besoins à l'aide de ces types de base [Tableau 2].

Valeurs	
flip::Bool	Valeur booléenne.
flip::Int	Nombre entier avec une représentation en 64 bits.
flip::Float	Nombre flottant avec une représentation en 64 bits.
flip::Enum	Énumération analogue à un type *enum* du langage C++.

[108] Nous faisons notamment référence à des solutions telles que *peerdata* (2.2.1.1, p. 67) ou encore *Purée Data* (2.2.2.2, p. 80).

[109] Les enjeux spécifiques liés au contrôle de la concurrence seront abordés dans la partie dédiée à l'espace de coordination du logiciel, plus spécifiquement à la partie 5.1.5 (p. 180) de ce document.

flip::String	Chaîne de caractères, analogue au type *std::string* de la bibliothèque standard du C++.
Conteneurs	
flip::Array	Séquence ordonnée d'éléments, analogue aux conteneurs *std::list* ou *std::vector* de la bibliothèque standard du C++.
flip::Collection	Liste non-ordonnée d'éléments, analogue au conteneur *std::set* de la bibliothèque standard du C++.
Référence	
flip::ObjectRef	Référence à un autre objet du même modèle de données. Cette classe peut être rapprochée du concept de pointeur en langage C ou C++.
Types non persistants	
flip::Message	Transmission d'un message avec des données non persistantes à tous les clients connectés au document.
flip::Signal	Transmission de messages locaux, non persistants, ni collaboratifs.

Tableau 2 – Présentation non-exhaustive des différents types de données de base fournis par la bibliothèque *flip* et utilisés pour la création du modèle de données d'un patch au sein de l'application Kiwi. La colonne de gauche correspond au nom de la classe C++ (précédée de l'espace de nommage propre à *flip*), la colonne de droite offre une courte description de leurs fonctionnalités.

Le tableau ci-dessus liste et apporte une description des types de base de l'API *flip* qui correspondent à ceux que nous avons utilisés dans le cadre de la création du modèle de données représentant un patch dans l'application Kiwi[110] [Tableau 2]. Ces types sont des classes C++ avec lesquelles il nous a été possible de composer les structures de données définissant la partie modèle du patch de Kiwi. Ils peuvent être classés en différentes catégories. On y trouve des types permettant le stockage de simples valeurs comme un nombre entier (*flip::Int*) ou flottant (*flip::Float*), ou encore une chaîne de caractères (*flip::String*). Mais aussi des conteneurs, qui peuvent être ordonnés ou non-ordonnés (*flip::Array* ou *flip::Collection*), ou encore un type représentant une référence à un autre type au sein du même document (*flip::ObjectRef*). Ces

[110] Cette liste n'est pas exhaustive, le lecteur pourra se rendre à l'adresse suivante pour une liste et une description plus complète : http://developer.irisate.com/documentation/reference/index.html [consulté le 11/09/18].

types sont persistants dans le sens où leur valeur sera stockée au sein du fichier lors de la sauvegarde du document. L'API de *flip* dispose aussi de types non persistants qui permettent la mise en place d'une communication entre le modèle et le reste de l'application (*flip::Signal*) ou encore entre plusieurs utilisateurs du même modèle de données (*flip::Message*[111]). Cette bibliothèque intègre aussi une classe C++ spécifique (*flip::Object*) qui permet au développeur de définir des structures de données personnalisées.

L'élaboration du modèle de données du patch au sein de Kiwi s'est résumée à définir les éléments qui le composent et la manière dont ces éléments devaient être représentés. La modélisation du patch au sein de Kiwi avec les types *flip* de base exposés plus haut est illustrée par la figure qui suit :

[111] Cette classe spécifique a notamment permis l'élaboration de l'objet *hub*, présenté dans la section 6.1 (p. 215) de ce document.

Figure 21 – Représentation simplifiée du modèle de données relatif à un document (un patch) au sein de l'application Kiwi, sous la forme de structure C++ modélisée grâce aux types de base fournis par la bibliothèque *flip* [Tableau 2]. Une des conventions veut que chaque type personnalisé au sein du modèle de données hérite du type de base *flip::Object* comme l'illustre ces différentes vignettes. La vignette 1 expose les membres relatifs au Patcher (la classe racine de notre document), la 2 expose ceux d'un objet de base d'un patch Kiwi, la 3 est relative aux liens, la 4 stocke des informations spécifiques à chaque utilisateur du patch, enfin la 5 des informations relatives à la vue graphique du patch.

Un patch peut être vu de manière simple comme une structure de données comportant un ensemble d'objets et de liens qui relient ces objets [Figure 21, 1]. Ces deux types d'élément sont stockés au sein d'un conteneur *flip::Array* afin de maintenir l'ordre dans lequel ils ont été ajoutés au sein du patch [Tableau 2].

Chaque objet au sein du modèle de données stocke un texte qui contiendra le nom de l'objet ainsi que les arguments optionnels tapés à la suite du nom. Ils comportent aussi des champs qui permettent de stocker des informations sur leur position ou encore leur taille au sein du patch [Figure 21, vignette 2, lignes 4 à 7], ainsi que le nombre et le type d'entrées et de sorties [Figure 21, vignette 2, lignes 8 et 9][112]. La classe *kiwi::Object* exposée ici est une classe générique. Chaque objet du langage Kiwi hérite de cette classe de base et de ses propriétés, mais peut aussi en ajouter pour ses besoins spécifiques[113].

Un lien est modélisé avec une référence vers deux objets du modèle correspondant à l'objet expéditeur et l'objet destinataire de la connexion ainsi qu'avec des index correspondants à l'index de l'entrée et de sortie des objets qu'il sert à relier [Figure 21, 3].

En plus de contenir des objets et des liens, la structure de données du Patcher possède aussi un conteneur spécifique permettant la sauvegarde d'informations spécifiques à chaque utilisateur du document [Figure 21, vignette 1, ligne 5]. Cette structure, nommée ici *PatcherUser*, ne stocke, dans la version actuelle de l'application, qu'une collection d'informations relatives à la vue d'un patch [Figure 21, vignette 4, ligne 3]. Les données relatives aux informations d'une vue d'un utilisateur au sein du patch sont la sélection des objets et des liens sous la forme d'une collection de référence à des éléments du même document (liens et objets) [Figure 21, vignette

[112] Les structures nommées *Inlet* et *Outlet* [Figure 21, vignette 2, lignes 8 et 9] ne sont pas exposées ici pour une commodité de lecture. Elle comporte simplement des informations relatives au type de données qu'accepte de recevoir ou d'envoyer un objet (message ou signal).

[113] La structure de données de l'objet *message* de Kiwi comporte par exemple un membre spécifique de type *flip::String* qui va servir à stocker le texte entré par l'utilisateur au sein de la boîte-message.

5, lignes 3 et 4], mais aussi le statut courant du patch (mode de jeu ou mode d'édition) ou encore le niveau de zoom souhaité [Figure 21, vignette 5, lignes 6 et 7][114].

Ce modèle de données peut être sauvegardé en tant que fichier par l'application cliente pour être rechargé ensuite. Dans ce cas l'application charge un document local qui n'est pas connecté au réseau, au même titre que l'on pourrait charger un fichier de type patch au sein des environnements Max ou Pure Data. La modélisation du patch a une influence cruciale dans la mesure où elle définit à la fois les informations qui seront sauvegardées au sein du document mais aussi les informations qui seront partagées entre les utilisateurs. Lors d'une session d'édition de patch collaborative, c'est cette structure même qui sera répliquée et partagée entre les différents utilisateurs, connectés cette fois-ci à un document distant sur un serveur, qui disposeront alors tous de cette même représentation du patch. À chaque modification du modèle, le système *flip* va synchroniser cette structure de données de façon à maintenir une cohérence des données sur chacun des sites distribués. Nous verrons cela plus en détails dans la sous-section 4.2.3, mais avant cela, concentrons-nous sur la manière dont est architecturée la partie cliente de l'application.

4.2.2. Architecture de l'application client

L'architecture sous-jacente d'un document (un patch), au sein de l'application cliente de Kiwi peut être rapprochée du patron de conception Modèle-Vue-Contrôleur (MVC), très populaire dans le domaine de la création logicielle. Dans le contexte de Kiwi, le modèle d'un patch correspond au document tel qu'il a été modélisé grâce à la bibliothèque *flip* et exposé

[114] Cette structure de données pourra à terme accueillir d'autres variables membres qui pourront servir à stocker par exemple des informations sur la position de la fenêtre, sa taille, ou encore la zone visualisée au sein du patch. Néanmoins ces développements n'ont pas encore pu avoir lieu par manque de temps durant le projet. Ils représentent donc pour nous une piste de développement future.

précédemment [Figure 21]. Le modèle sert uniquement au stockage des informations, il contient par exemple l'information qu'un objet (au sens du modèle) comporte le texte « noise~ » mais ignore complètement que cet objet sert par exemple à produire un bruit blanc lorsqu'il est instancié dans le contexte d'un patch, c'est l'interprétation de ce modèle qui va fournir ce type de logique.

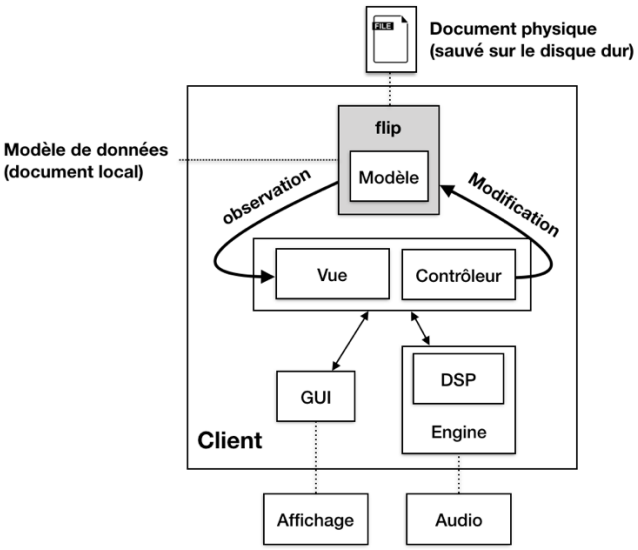

Figure 22 - Schéma représentant l'architecture cliente de l'application Kiwi chez un utilisateur chargeant un document (un patch) en local depuis sa machine. L'application interprète le modèle de données pour fournir un rendu graphique (GUI), logique (Engine) et audio (DSP) de celui-ci à l'utilisateur au sein de l'application.

La partie *Contrôleur* est celle qui va opérer des changements sur ce modèle, par exemple ajouter un objet au sein du patch ou ajouter une connexion entre deux objets. La modification du document se fait sur le document *flip*[115] par le biais d'une *transaction* qui représente une différence entre deux états du document. A chaque modification de la structure du modèle, le système génère une notification à la partie *Vue* qui va observer ces changements. La *Vue* dans ce genre de patron de conception n'est pas à prendre dans une acception purement graphique,

[115] http://developer.irisate.com/documentation/reference/Document.html [consulté le 15/09/18].

c'est une partie du code qui réagit simplement aux changements ayant été effectués sur le modèle. Elle peut alors servir à mettre à jour l'interface graphique, la manière dont est visualisé un patch, mais aussi à mettre à jour la structure logique du patch (*engine*), ou encore son rendu sonore (*DSP*). Cette architecture est illustrée par la [Figure 22].

La partie *Vue* et la partie *Contrôleur* ont la particularité d'être couplés dans l'architecture telle qu'elle est actuellement en place dans Kiwi [Figure 22]. Nous pouvons distinguer trois sortes de *Vues*-*Contrôleur* du même modèle de données au sein de Kiwi :

- Celle relative à son affichage graphique, que voit l'utilisateur au sein de la fenêtre Patcher du logiciel et avec laquelle il peut interagir (*GUI*).
- Celle relative à la logique des messages transmis entre les objets lors de l'exécution du patch (*engine*).
- Et enfin celle relative au rendu sonore du patch (*DSP*).

Actuellement, la partie *DSP* est contenue et contrôlée au sein de l'*engine* [Figure 22]. Ces deux *Vues* interprètent et modifient le modèle à leur façon et suivant leurs besoins spécifiques à chaque modification qui est opérée sur le document. L'*engine*, contrairement à la partie *GUI*, n'a par exemple pas besoin de connaître en détail l'état des sélections au sein du patch. Elle va en revanche observer l'ajout et la suppression d'objets et de liens pour déterminer la logique de transmission des messages et savoir quel objet logique instancier. Lorsqu'elle reçoit la notification qu'un nouvel objet a été ajouté au document, elle va lire le texte contenu dans la variable du modèle [Figure 21, vignette 2, lignes 3] puis l'interpréter en créant un objet approprié. Si le texte est « + 10 », elle se chargera de créer un objet ayant pour fonction d'ajouter la valeur dix au nombre reçu en entrée. Si le texte est « noise~ », l'*engine* transmet cette information à la partie *DSP* qui va s'en servir à son tour pour mettre à jour la chaîne de

traitement audio[116]. La partie *GUI* quant à elle ne s'intéresse pas à la logique interne de l'objet, en revanche, elle va instancier des composants graphiques spécifiques au sein du patch en fonction du nom de l'objet contenu dans le modèle ou encore faire le choix d'afficher différemment un lien si celui-ci représente une connexion de type message ou signal. Notons enfin que ces deux parties contrôlant et observant le modèle peuvent communiquer entre-elles par l'intermédiaire du modèle grâce au type *flip::Signal* de l'API *flip*, conçu spécifiquement pour cet usage [Tableau 2].

Ce que l'on peut retenir de ces aspects de conception un peu techniques est que la partie modèle, celle qui stocke la structure du document, est séparée de la logique du patch, de son rendu sonore et graphique. En cela, l'architecture diffère de celle de Pure Data, [Figure 11, p. 67], dans laquelle seule la partie *Pd-GUI* est séparée de la partie logique (*Pd-engine*), cette dernière ayant aussi le rôle de *Modèle* au sein de l'application. Lorsque deux utilisateurs sont connectés au même document, ils doivent disposer du même modèle de données, en revanche la manière de rendre ce modèle de données, c'est-à-dire de l'interpréter graphiquement ou de manière logique et sonore doit être gérée par chacune des applications clientes. C'est donc dans un contexte collaboratif que cette architecture prend tout son sens et c'est aussi là tout l'intérêt d'avoir séparé la partie *Modèle* du reste de l'application, comme on pourra l'observer dans la sous-section suivante.

4.2.3. Architecture client-serveur

Dans un contexte collaboratif, lorsque plusieurs utilisateurs éditent en même temps le même document, l'architecture cliente se comporte globalement de la même manière que lors d'un

[116] Notons qu'à chaque changement du modèle entrainant une modification de la chaîne *DSP* (ajout ou suppression d'un lien ou d'un objet de type signal), celle-ci est recompilée pour se mettre à jour automatiquement.

contexte d'édition mono-utilisateur. Ce qui change néanmoins, c'est que le document local n'est plus le document de référence, il représente une copie, une réplique d'un document de référence situé sur un serveur distant auquel chaque application cliente est connectée. Le serveur maintient ce document de référence, qui contient le même modèle de données représentant un patch que nous avons déjà évoqué [Figure 21]. En revanche, toute la partie dédiée à l'interprétation de ce modèle, que l'on peut trouver au sein de l'application client, est absente du modèle.

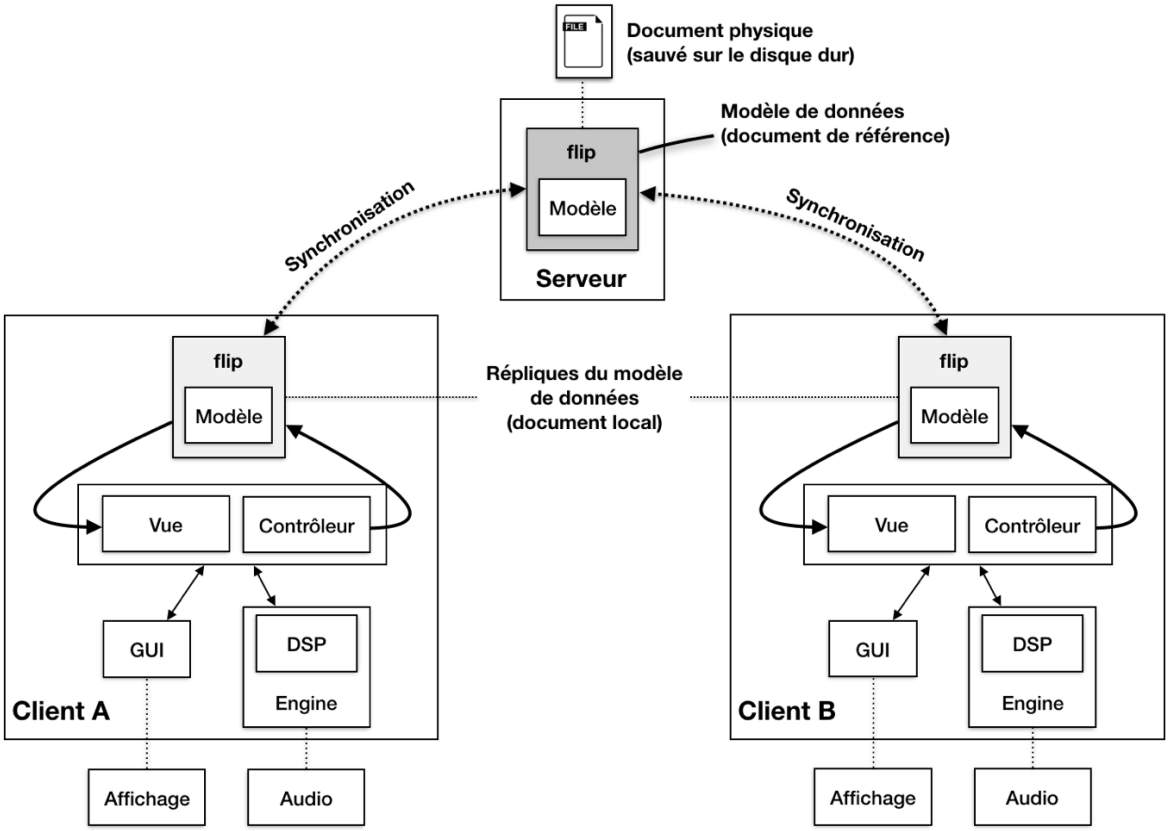

Figure 23 - Schéma décrivant de manière simplifiée l'architecture client-serveur de Kiwi lorsque deux utilisateurs sont *connectés* au même document (patch) sur un serveur central. Le serveur se charge de lire et sauvegarder le document physique sur le disque et de synchroniser les répliques locales du modèle de données (document virtuel) entre chaque client. Les applications clientes interprètent le modèle de données pour fournir un rendu graphique (GUI), logique (Engine) et audio (DSP) de celui-ci à l'utilisateur au sein de l'application locale.

Cette architecture client-serveur est illustrée par le schéma ci-dessus [Figure 23] qui expose la topologie de réseau d'une connexion à un même document via un serveur par deux applications clientes. Chaque modification au modèle continue à être effectuée sur la version locale du document, de cette façon, l'édition peut se faire aussi rapidement que dans un contexte mono-utilisateur, permettant alors de maintenir un temps de réponse minimal requis dans le cadre d'une application collaborative synchrone. Chaque modification est ensuite transmise au serveur sous la forme d'une transaction pour mettre à jour le document de référence sur le serveur qui se charge de synchroniser l'ensemble des autres répliques locales du même document chez tous les utilisateurs connectés [Figure 23]. Lorsque l'application locale est notifiée d'une modification du document, que celle-ci ait été réalisée par l'utilisateur local ou un autre utilisateur sur le réseau, elle met à jour ses *Vues* pour refléter les changements[117].

Il est intéressant de comparer l'architecture réseau de Kiwi, permettant la réplique des données entre les applications clientes, de celle de *netpd*[118]. Dans le cadre de la solution *netpd*, les documents physiques ne sont pas situés sur un serveur, comme ici, mais sur chacun des sites distribués, posant alors des problèmes potentiels de synchronisation des données et d'accessibilité à ces ressources. D'autre part, c'est l'ensemble des patchs *netpd* qui est partagé au cours d'une session de jeu connectée au serveur. Ici, tous les documents sont indépendants. Les utilisateurs peuvent se connecter aux documents de leur choix, accessibles directement sur le serveur et donc depuis n'importe quelle machine et à n'importe quel moment. Ces

[117] Notons qu'une étape cruciale à ce processus a été omise dans ce cadre. Il arrive que des conflits apparaissent lors d'une édition concurrente du document et que les transactions envoyées soient invalides. Dans ce cas, la transaction est rejetée par le serveur. Ce problème, qui est en fait celui du contrôle de la concurrence, sera évoqué plus en détail dans la sous-section 5.1.5 (p. 180).

[118] Présentée dans la première partie de ce mémoire (2.2.2.2, p. 80) et illustrée par la [Figure 16, p. 82].

fonctionnalités faisant partie de l'espace de coordination seront évoquées plus en détail par la suite[119].

4.3. Opérations

Nous avons donné un aperçu général du langage Patcher de Kiwi, du patch en tant que document, à travers l'étude de sa modélisation, ainsi que de l'architecture générale de la solution logicielle. Nous souhaitons à présent revenir sur la représentation graphique de la solution et sur les opérations offertes à l'utilisateur dans un contexte d'utilisation local ou collaboratif. Pour cela, nous présenterons la mise en œuvre de l'interface graphique du composant Patcher ainsi que les différentes fonctions qu'il propose à l'utilisateur. Nous verrons alors que les aspects architecturaux et relatifs à la conception du modèle que nous avons exposés plus haut ont une incidence directe sur ces dernières.

4.3.1. Mode de *jeu* et mode d'*édition* du patch

L'interface graphique du composant Patcher fait intervenir, tout comme au sein des environnements de patching traditionnels de type Pure Data ou Max, deux modes différents d'interaction qui permettent alternativement d'éditer ou de contrôler les éléments du patch. La capture d'écran ci-dessous présente un patch dans lequel un procédé de traitement de synthèse audio, fondé sur de la modulation de fréquence, est mis en œuvre [Figure 24]. Cette figure expose les deux modes d'interaction possibles avec le patch. Ces deux modes sont différenciables graphiquement par l'icône de cadenas ouvert ou fermé présente sur la barre d'outil de la fenêtre détaillée un peu plus loin, mais aussi grâce au texte présent sur la barre de titre.

[119] Voir notamment section 5.2 (p. 190) de ce document.

On pourra noter que le patch, en mode *jeu*, présente moins d'informations graphiques qu'en mode édition. Les entrées et sorties des objets sont cachées et certains objets, tels que les commentaires, adaptent leur apparence afin d'alléger l'interface pour fournir une meilleure lisibilité au traitement.

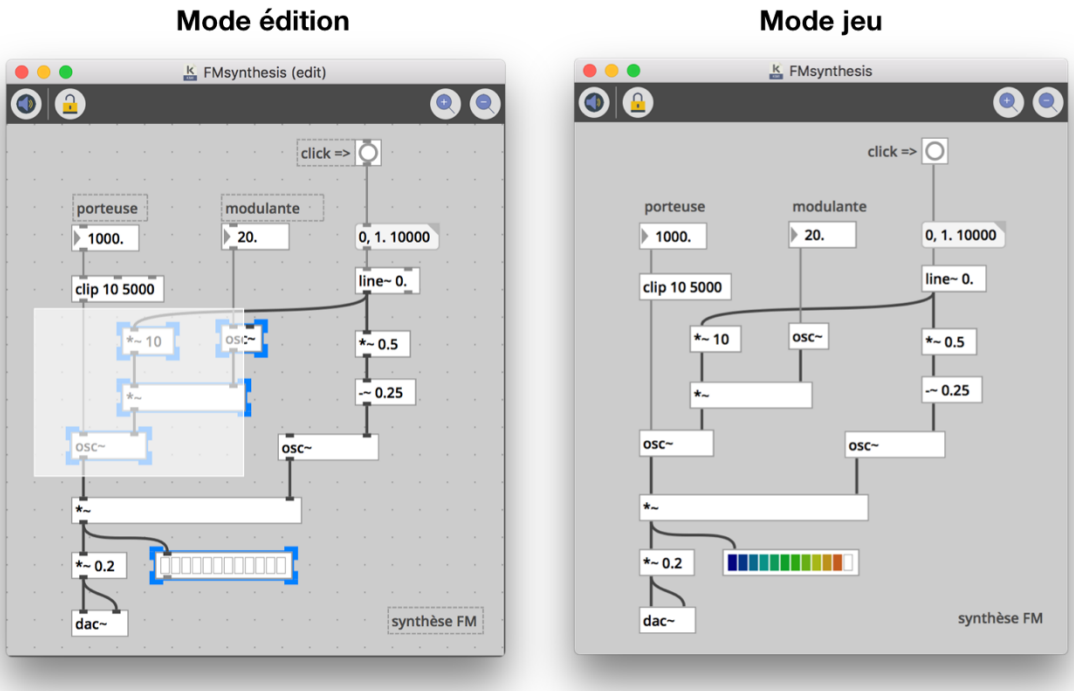

Figure 24 – Capture d'écran d'un même patch Kiwi dans un contexte mono-utilisateur, exposant les deux modes d'interaction possibles avec le patch. Le patch sur la gauche représente le mode d'édition, les objets sélectionnés sont représentés avec un contour bleu et le rectangle correspond à une sélection au *lasso* en train d'être réalisée par l'utilisateur. Sur la droite, le patch est en mode jeu, l'audio est allumé et le patch est verrouillé.

Lorsque le patch est en mode *édition* [Figure 24, à gauche], il supporte les opérations standard d'un logiciel de patching à savoir, l'ajout, l'édition ou la suppression d'objets et de liens, ou encore des fonctions qui permettent de copier, coller, couper, remplacer ou dupliquer un objet ou un groupe d'objets avec ses liens. Les objets peuvent aussi être édités, redimensionnées ou repositionnées au sein du patch. Lors d'une édition collaborative de document, ce sont ces

opérations d'édition qui sont partagées avec les autres participants. Lorsque deux personnes (ou plus), sont connectées au même document, toutes leurs actions d'édition sont appliquées sur leur document local, qui produit alors une transaction, qui est ensuite transmise au serveur puis aux autres utilisateurs connectés pour qu'ils l'appliquent à leur tour au sein de leur propre document local. Ceci assure alors le maintien de la synchronisation entre chacune des versions locales.

L'utilisabilité du logiciel est renforcée dans la mesure où toutes ces actions peuvent être effectuées grâce à des commandes spécifiques accessibles avec des raccourcis clavier. Ces opérations s'effectuent en sélectionnant au préalable les objets en cliquant dessus directement ou par l'intermédiaire d'un lasso qui permet d'effectuer une sélection groupée, comme montré à la [Figure 24, à gauche]. La sélection des objets au sein du patch a une double fonction dans le cadre d'une utilisation mono-utilisateur, elle permet évidemment de notifier graphiquement la sélection grâce à l'apparition d'un rectangle bleu, mais ce rectangle est aussi actif et expose des poignées permettant le redimensionnement des boîtes-objets. Nous verrons que, lors d'une session collaborative, les sélections ont aussi une troisième fonction, celle de notifier l'activité des autres participants sur le patch[120].

Toutes ces actions d'édition peuvent être annulées ou restaurées grâce à un système d'*undo/redo* infini. La mise en œuvre de ce mécanisme au sein de l'interface du Patcher est soutenue au niveau logiciel par la bibliothèque *flip* grâce à un système d'historique permettant le stockage des transactions en local. Si la mise en œuvre ressemble à ce que l'on peut trouver

[120] Ce point sera abordé dans la sous-section 5.1.2 de ce document (p. 166).

dans d'autres applications de patching, comme nous l'avons dit dans la première partie de ce mémoire et comme nous le verrons dans le prochain chapitre, ce système a dû s'adapter aux spécificités de l'édition multi-utilisateur[121].

Le mode de *jeu*, permet quant à lui le contrôle du patch par l'intermédiaire d'objets graphiques, qui offrent des capacités d'interaction directe avec le traitement sonore ou le processus de calcul via des entrées physiques de la machine telles que la souris ou le clavier. Dans ce cas précis de modulation de fréquence, l'utilisateur peut contrôler la valeur de la fréquence porteuse et de la modulante à travers les boîtes-nombres ou encore déclencher la génération d'une enveloppe d'amplitude en appuyant sur le bouton prévu à cet effet [Figure 24, à droite]. Ces opérations de jeu, contrairement à celles d'édition, ne font pas appel à une modification du modèle, et font seulement intervenir les composants logiciels de vue graphique (*GUI*) et logique (*engine*) [Figure 22, p. 133]. Elles ne sont donc pas partagées lors d'une session collaborative et restent donc locales[122].

Comme nous l'avons vu lors de la présentation de la modélisation du patch, le mode d'interaction courant (jeu ou édition), est compris dans la structure du modèle de données du patch. À ce titre, il est donc sauvegardé au sein du document de manière persistante de façon à ce que l'utilisateur puisse retrouver ce réglage lorsqu'il charge à nouveau un document. En revanche, la manière dont cette variable a été modélisé au sein du modèle permet d'offrir un réglage contextuel qui permet aux utilisateurs d'être indépendants [Figure 21, vignette 1, 4 et 5, vignette 5 ligne 6]. Dans un contexte collaboratif, un utilisateur peut donc très bien être en mode jeu pendant qu'un autre est en mode édition sur le même patch dans la mesure où la

[121] Ce point sera abordé dans la sous-section 5.1.4 de ce document (p. 178).

[122] Ce point sera abordé plus en détail dans la sous-section 4.3.5 (p. 150) de ce document.

variable du modèle est spécifique à une vue d'un utilisateur au sein du patch (de la même façon deux vues locales du même patch peuvent aussi être différentes).

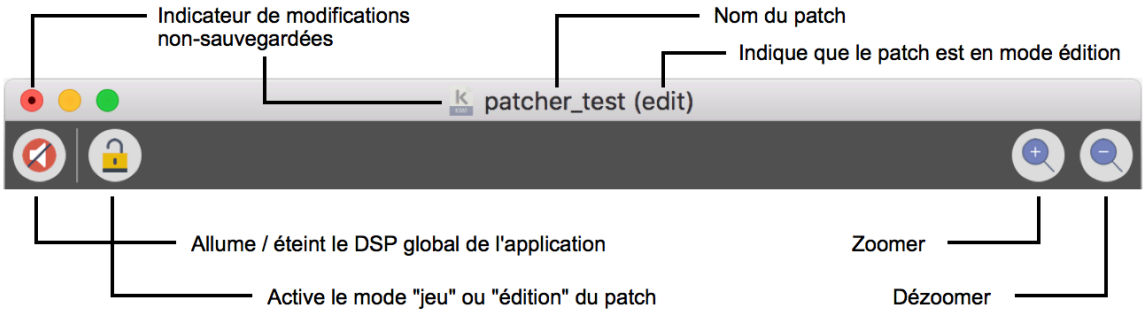

Figure 25 – Capture d'écran présentant la barre de titre et la barre d'options d'un patch local avec une légende expliquant le rôle des différents boutons qui la compose.

La barre d'options se situant en haut de la fenêtre Patcher contient des boutons permettant d'allumer ou d'éteindre le traitement audio général de l'application, de passer le patch courant en mode jeu ou en mode édition ou encore d'agrandir ou de rétrécir la zone visible du patch. Toutes ces actions n'affectent que l'utilisateur local, même lors d'une session collaborative. La barre de titre est quant à elle dépendante du système d'exploitation sur lequel l'application est exécutée. Sur la plateforme *MacOS*, des indicateurs sont présents pour signifier à l'utilisateur que le patch comporte des modifications non-sauvegardées lorsque celui-ci est rattaché à un fichier local [Figure 25]. Nous verrons dans le chapitre consacré à l'espace de coordination, que l'aspect et le comportement de cette barre d'outil changent légèrement lorsque le patch est connecté à un document distant et utilisé dans un contexte collaboratif[123].

4.3.2. Instanciation des objets dans un patch

Comme dans tout langage de programmation, un apprentissage de sa syntaxe, sa grammaire et de son vocabulaire est requis si l'on veut exploiter tout son potentiel. Le cas des langages de

[123] Ce point sera abordé à la sous-section 5.1.1 (p. 164) de ce document.

programmation visuelle ne fait pas exception. Plus spécifiquement dans le langage Patcher, la principale difficulté réside, en particulier pour des utilisateurs débutants, dans le fait de devoir apprendre le nom des objets qui peuvent être instanciés ou non au sein d'un patch.

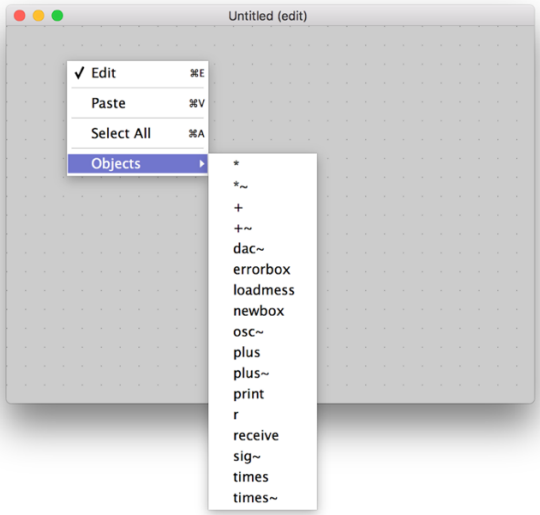

Figure 26 – Capture d'écran de la fenêtre *Patcher* de l'application Kiwi (version v0.0.3) exposant un menu contenant la liste exhaustive des objets instanciables au sein d'un patch.

Dans la première version de l'application nous avions donc mis en œuvre un système simple consistant à présenter à l'utilisateur la liste exhaustive des objets disponibles au sein d'un menu contextuel accessible en effectuant un clic-droit sur le patch [Figure 26]. Ce système se rapprochait du menu global *Add* de Pure Data à partir duquel il est possible d'instancier certains objets, notamment graphiques. Il convenait assez bien lorsque le nombre d'objets était limité mais, la liste des objets supportés par Kiwi grandissant petit à petit, il nous a fallu trouver un autre système améliorant l'expérience utilisateur. Pour plus de facilité dans la recherche des objets, nous avons donc décidé de mettre en œuvre un système d'auto-complétion à partir de la *v0.1* de Kiwi qui convient mieux à ce genre d'usage.

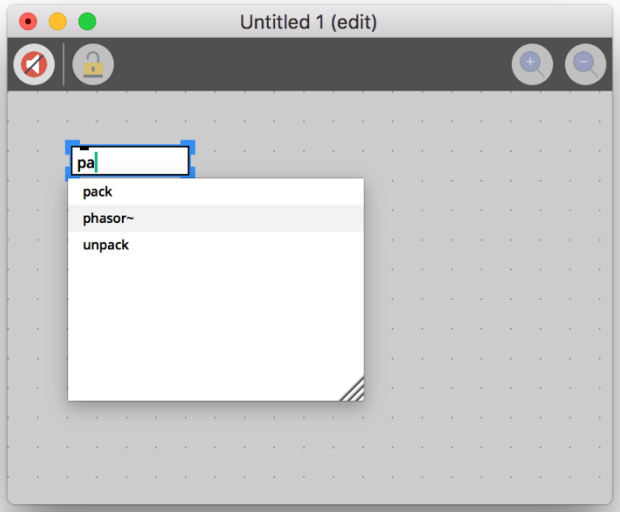

Figure 27 – Capture d'écran de la fenêtre *Patcher* de la dernière version de l'application Kiwi exposant la fenêtre contextuelle apparaissant en dessous des boîtes-objets lorsque les utilisateurs éditent un texte à l'intérieur.

Quand l'utilisateur commence à taper du texte au sein d'une boîte-objet dans un patch, une fenêtre contextuelle apparaît désormais en dessous de celle-ci, proposant une liste d'objets instanciables en fonction des caractères qui ont été entrés [Figure 27]. Ce système d'auto-complétion repose sur un code de logique floue qui permet de filtrer et de trier la liste des objets disponibles à partir de leur nom en leur attribuant à chacun une pondération qui dépend du texte entré par l'utilisateur et de certains paramètres fixés à l'avance (bonus pour les caractères consécutifs, pénalité pour les caractères non-présents etc.)[124].

[124] Le code sous-jacent à cette mise en œuvre est consultable à l'adresse suivante : https://github.com/Musicoll/Kiwi/blob/d8fb17231c41d59d139be012138dd61bd4fb0b38/Client/Source/KiwiApp_Utils/KiwiApp_SuggestList.h [consulté le 06/09/18]. Ce code est une adaptation du code de Forrest Smith décrit dans l'article suivant : https://blog.forrestthewoods.com/reverse-engineering-sublime-text-s-fuzzy-match-4cffeed33fdb [consulté le 06/09/18].

Dans l'exemple présenté ci-dessus [Figure 27], où l'utilisateur a commencé à entrer le texte "pa", l'objet *pack* est proposé en premier dans la liste car c'est celui qui correspond le plus au texte qui a été entré, l'objet *phasor~* est proposé en second car il contient tous les caractères et qu'il commence par le même caractère que le texte qui a été entré, l'objet *unpack* est présenté en troisième dans la mesure où, même s'il ne commence pas par les mêmes caractères, le texte entré est contenu entièrement dans le nom de cet objet. La liste exhaustive des objets est disponible en pressant la touche *echap* du clavier au sein d'une boîte en train d'être éditée ou si aucun texte n'a encore été tapé. Notons que dans un contexte collaboratif, l'activité d'édition d'une boîte-objet n'est pas partagée directement. La modification de l'objet n'est partagée avec les autres que lorsque l'utilisateur met fin à l'action d'édition.

4.3.3. Raccordement des objets

Pour relier deux objets entre eux et ainsi permettre une communication du premier objet vers le second, l'utilisateur peut créer un lien en cliquant sur une sortie puis en glissant la souris vers une des entrées de l'objet à raccorder[125]. Néanmoins, seules les entrées/sorties compatibles peuvent être raccordées entre-elles. Une sortie de type signal ne peut être reliée par exemple qu'à une entrée acceptant aussi du signal.

La nature d'un lien est déterminée par le type de sortie de l'objet auquel il est raccordé. Si la sortie est de type signal, le lien modélise une communication continue de type signal entre les deux objets ; sinon, celui-ci modélise une communication discrète sous la forme de messages. La création d'un lien est facilitée à travers l'interface graphique par une proposition de

[125] Il est aussi possible d'effectuer cette opération dans le sens inverse, en partant de l'entrée d'un objet et en créant un lien vers la sortie d'un objet.

connexion contextuelle automatique en fonction du type de données accepté par l'entrée/sortie à partir de laquelle a été initiée celle-ci.

La manière de représenter les liens ainsi que leur état de sélection a évoluée au fil des versions et a été revue à plusieurs reprises. À partir de la première version ayant fait intervenir du traitement du signal au sein de Kiwi [Paris & al., 2017], nous avons choisi de colorer les liens faisant transiter des messages en noir, et les liens représentant une transmission du signal en vert [Figure 28, à gauche].

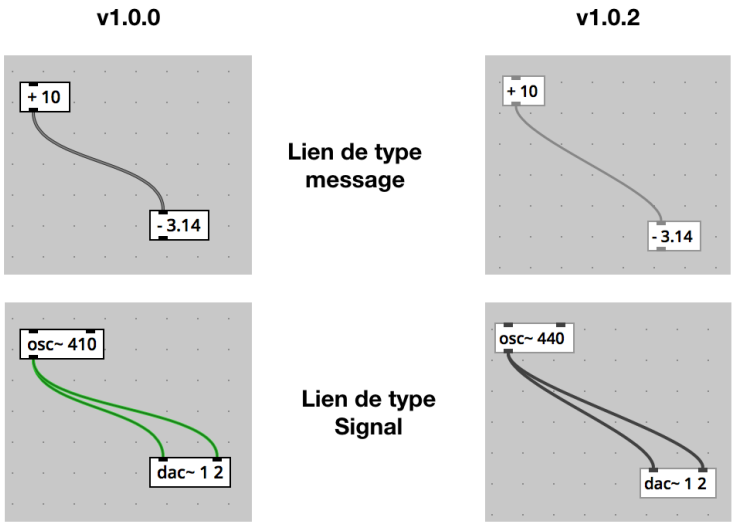

Figure 28 – Capture d'écran montrant les deux types de liens pouvant relier les objets à la version *v1.0.0* (à gauche) et à la v1.0.2 (à droite) du logiciel Kiwi.

Cette distinction fondée sur la couleur fonctionnait assez bien pour les utilisateurs qui différenciaient les types de connexions. En revanche, elle apportait aussi de la confusion quand il s'agissait de représenter son état de sélection, en particulier dans un contexte collaboratif, comme nous le verrons par la suite (la couleur originale verte du lien se confondant avec celle

de la sélection)[126]. Ce système a été revu dans la version *v1.0.2* [Figure 28, à droite] en s'inspirant notamment graphiquement de la mise en œuvre de *Purr Data*. Les liens de type signal et de type contrôle se distinguent désormais grâce à une nuance de gris et une différence d'épaisseur, ce qui nous donne l'opportunité d'exploiter sans problèmes les couleurs pour représenter l'état de leur sélection. Notons que dans un contexte d'édition collaborative, tout comme l'activité d'édition d'une boîte-objet, les liens ne sont partagés qu'une fois qu'ils ont été reliés à deux objets au sein du patch.

4.3.4. Contexte local de visualisation du patch

L'interface du Patcher est un composant graphique dont la taille varie dynamiquement en fonction de la dimension de la fenêtre de visualisation dans laquelle il est chargé, mais aussi de la position et de la taille des objets qu'il contient. La position des objets au sein du patch, telle qu'elle est modélisée dans le document [Figure 21, p. 130, vignette 2, ligne 4 et 5], est relative à l'origine du patch. Contrairement à Pure Data ou *Purr Data* qui ne font pas la distinction entre la zone négative et positive d'un patch, nous avons choisi de reprendre dans Kiwi la solution graphique employée par Max en différenciant les deux zones à l'aide d'un léger contraste [Figure 29]. Lorsque la taille du patch excède celle de la zone de visualisation, des barres de défilement horizontales et verticales apparaissent en bas et à gauche de la fenêtre pour permettre à l'utilisateur de se déplacer à l'intérieur de celle-ci.

[126] Lorsque plusieurs personnes sont connectées au même patch dans Kiwi, chaque utilisateur peut visualiser les sélections des autres utilisateurs, liens compris en temps réel. Ce point sera abordé dans la sous-section 5.1.2 de ce document (p. 166).

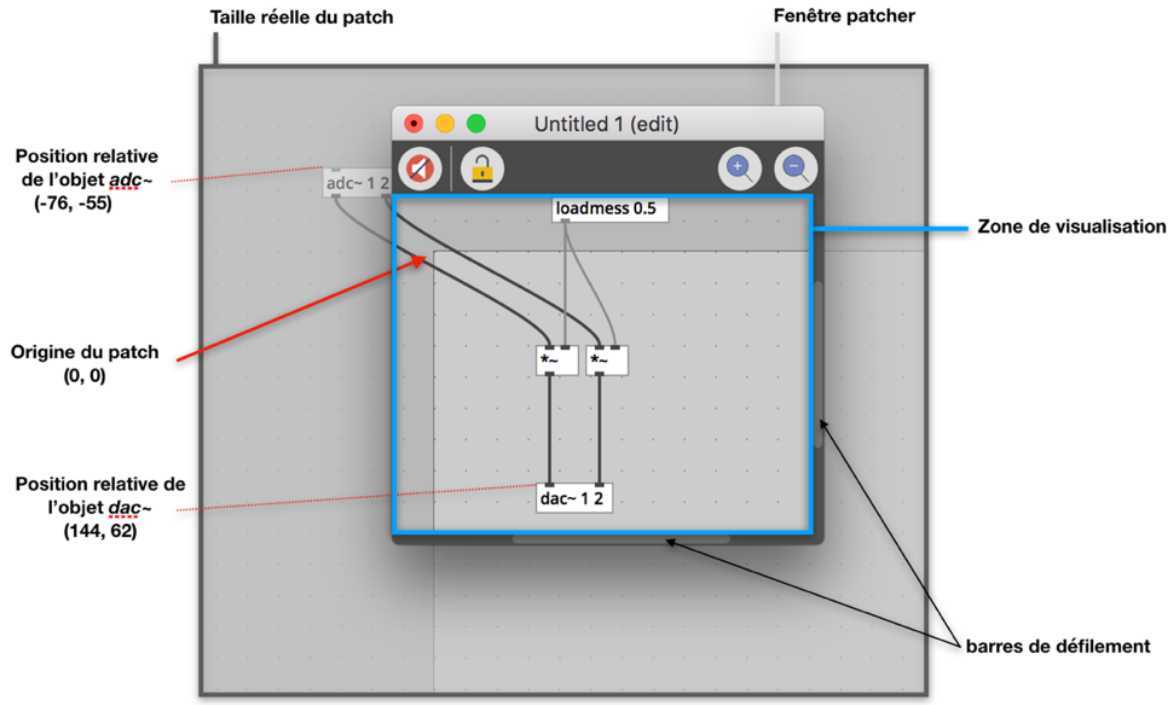

Figure 29 – Illustration du système de *viewport* dans Kiwi. En arrière-plan on peut apercevoir le patch dans son ensemble. Au premier-plan est représenté la fenêtre Patcher qui accueille un composant graphique de type *viewport* qui permet de naviguer à l'aide de barres de défilement horizontales et verticales au sein du patch pour en offrir une vue réduite.

Dans la mesure où la vue graphique d'un patch est rendue localement, grâce à l'architecture répliquée de Kiwi, dans un contexte multi-utilisateur chacun peut placer où il le souhaite les fenêtres et visualiser une zone spécifique du patch sans que cela n'affecte les autres utilisateurs. En ce sens nous sommes donc déjà en présence d'un exemple de relâchement du principe de *WYSIWIS*, donc de *WYSIAWIS*, tel que défini dans la première partie de ce mémoire[127].

Dans Kiwi il est aussi possible de visualiser le même patch dans plusieurs fenêtres en même temps. Cette fonctionnalité de multi-vue s'inspire de celle mise en œuvre dans le logiciel Max.

[127] Voir section 3.7 (p. 99) de ce document.

Elle peut être utilisée pour visualiser localement plusieurs parties spécifiques d'un même patch tout en gardant une vue globale de l'ensemble du patch dans une fenêtre séparée.

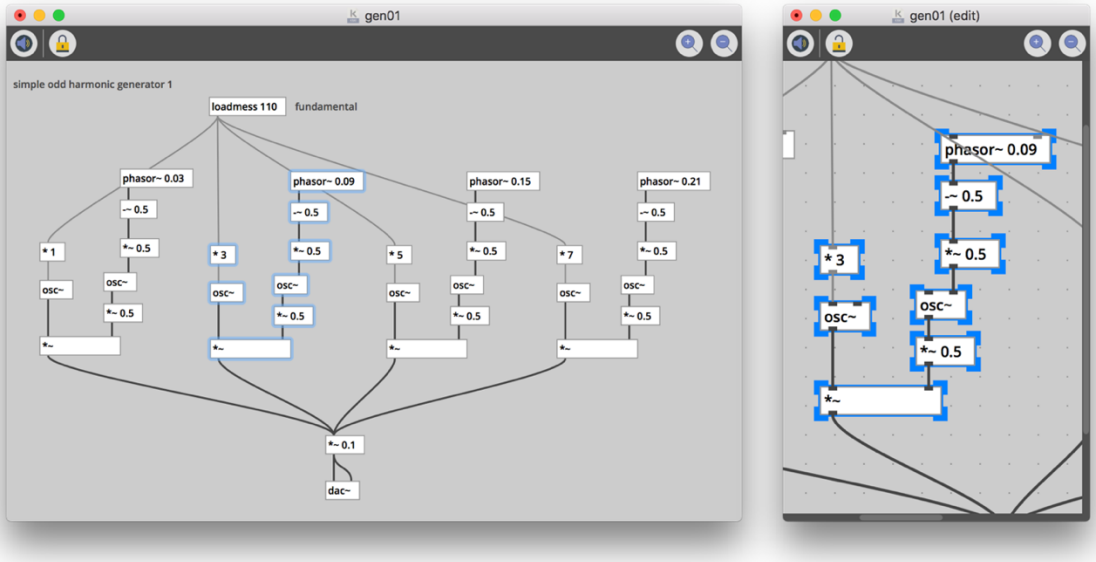

Figure 30 – Capture d'écran montrant deux vues différentes d'un même patch de synthèse audio chez un utilisateur. La vue de gauche correspond à une vue verrouillée et rétrécie, la vue de droite est une vue grossie du même patch en mode édition. La sélection des objets, réalisée depuis la vue de droite, est visible sur la vue de gauche grâce à une couleur de contour bleue donnée aux objets.

Dans le patch présenté ci-dessus, un traitement de synthèse additive fondé sur la production d'harmoniques impaires est opéré. La fenêtre de gauche permet de visualiser l'ensemble du traitement mis en œuvre grâce à une vue rétrécie du patch. À droite, une seconde fenêtre présente une vue grossie de ce même patch dans laquelle les objets nécessaires à la fabrication du second partiel sont sélectionnés [Figure 30].

Les sélections effectuées sur une vue sont affichées au sein des autres vues du patch avec une couleur légèrement contrastée. Le système de multi-vue associé au système de sélection peut se révéler particulièrement utile par exemple dans le cadre d'un cours, pour exposer facilement une partie locale d'un patch afin d'en expliquer son fonctionnement, tout en maintenant une

vue d'ensemble dans une fenêtre à part, afin de situer l'opération locale dans son contexte global. L'utilisabilité de cette fonctionnalité est renforcée par le comportement du zoom qui s'adapte au contexte de sélection du patch. Pour offrir un focus sur un groupe d'objet l'utilisateur peut exercer une sélection puis un zoom, la zone de visualisation se recentrera alors automatiquement sur la zone de sélection. Notons que la valeur de zoom, telle qu'elle est modélisée [Figure 21, p. 130, vignette 1 et 4, et vignette 5, ligne 7], est stockée au sein du document et peut donc être restaurée au chargement du patch, néanmoins celle-ci reste spécifique à chaque utilisateur qui peut contrôler ce réglage sans affecter celui des autres dans le contexte d'une session collaborative.

La manière dont l'architecture de l'application cliente est conçue et dont la manière dont a été modélisé le patch a une influence sur la vue du patch entre les utilisateurs connectés au même document mais aussi sur son exécution.

4.3.5. Contexte local d'exécution du patch

La figure ci-dessous expose la même version d'un patch exécuté chez deux utilisateurs connectés en temps réel au même document sur un serveur [Figure 31]. Le traitement de synthèse mis en œuvre ici est volontairement très simple pour des raisons didactiques. Il se compose d'un oscillateur à une fréquence de 440Hz (1) qui est modulé par une enveloppe d'amplitude (3) pouvant être déclenchée par un bouton (2). Le gain général de sortie (7) peut être géré à l'aide d'un potentiomètre linéaire (5) dont la valeur est envoyée à une boite-message pour définir la rampe de l'objet line~ (6) et lisser les échantillons.

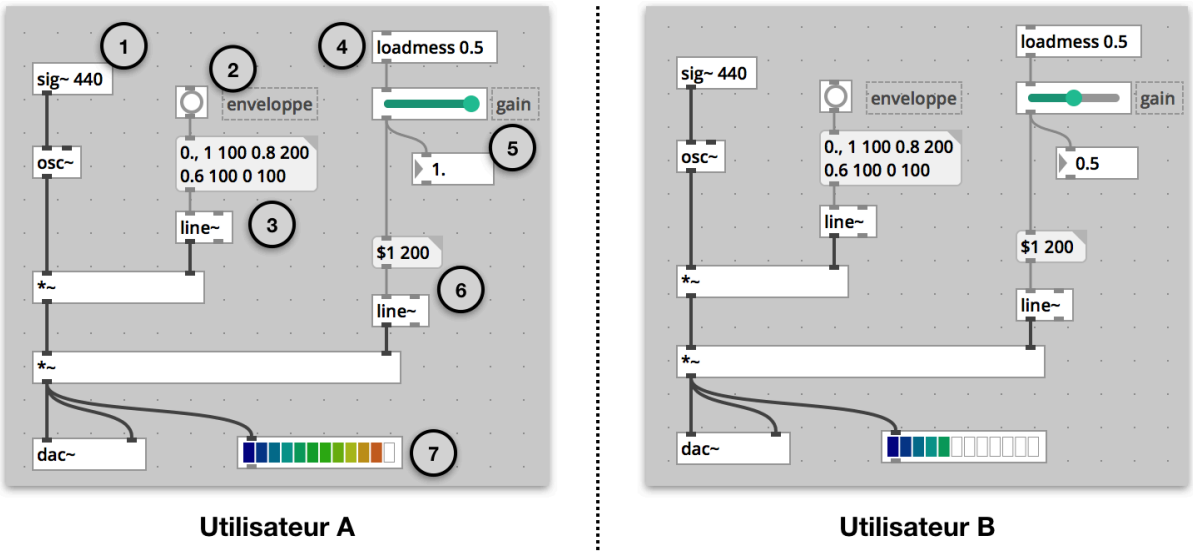

Figure 31 – Capture d'écran d'un même patch Kiwi tel qu'il apparaît chez deux utilisateurs différents, connectés en temps réel au même document. Les numéros présents sur le patch de l'utilisateur A sont repris dans le corps du texte qui suit. Nous nous réfèrerons par exemple à l'objet *sig~* en utilisant (1).

Cet exemple va néanmoins nous permettre de mieux comprendre ce que les utilisateurs partagent lorsqu'ils sont connectés à un document commun et ce qui reste local, c'est-à-dire ce qui n'est pas partagé avec les autres participants connectés. Comme on peut le constater, la structure du patch est la même chez les deux utilisateurs, les objets sont placés au même endroit sur le patch et sont raccordés de la même façon entre eux. Si un des utilisateurs fait une modification du patch telle que l'ajout d'un objet ou le raccordement d'un objet à un autre au sein du patch, celle-ci sera automatiquement répliquée chez le second utilisateur. Néanmoins tout n'est pas partagé lors de l'exécution de ce patch, comme peut le laisser transparaître l'interface graphique illustrée ici. La première chose que l'on peut voir, est que le signal de sortie n'est pas le même chez les deux utilisateurs. Ceci est dû au fait que le traitement de l'audio n'est pas centralisé dans un contexte collaboratif, comme nous l'avons expliqué lors de la présentation de l'architecture logicielle [Figure 22, p. 133] et [Figure 23, p.136], mais s'effectue sur chaque machine exécutant le patch à partir des données répliquées du modèle. La valeur interne du potentiomètre (1) est contenue dans la partie logique et n'est pas modélisée

au sein du document. Elle n'est donc pas répliquée entre les utilisateurs qui peuvent avoir des réglages qui leurs sont spécifiques, comme c'est le cas ici. Le patch, tel qu'il est conçu, définit une valeur de gain par défaut initialisée au chargement du patch par l'objet *loadmess* (4), l'utilisateur B dispose de cette valeur. En revanche, l'utilisateur A l'a modifiée en jouant avec l'interface de contrôle du gain (5), néanmoins, cette valeur n'a pas été transmise à l'autre participant du fait qu'elle n'appartient pas au modèle mais à la logique locale du patch. De cette façon, le contrôle du patch peut rester local au sein d'un patch exécuté dans l'environnement Kiwi. De la même façon, chaque utilisateur peut générer à loisir une nouvelle enveloppe d'amplitude au sein de ce patch en appuyant sur le bouton prévu à cet effet (2) sans que cela n'impacte l'autre utilisateur. Pour résumer on peut dire que ce que partagent les utilisateurs au sein du patch est la structure du traitement, son algorithmique propre. Ils partagent le fait que l'oscillateur soit réglé à une fréquence fixe (1), que la valeur temporelle de la rampe lissant le gain du signal de sortie soit de 200 millisecondes, dans la mesure où cette valeur est partagée au sein de la boîte-message (6). Ou encore la définition de l'enveloppe d'amplitude (3), qui sera la même pour tout le monde du fait qu'elle ait été écrite elle aussi au sein d'une boîte-message dont le texte fait partie du modèle de données et est donc à ce titre synchronisé entre chaque utilisateur. En revanche, le contrôle des paramètres exposés au sein du patch via les objets graphiques reste quant à lui local. Un utilisateur peut donc décider du niveau de sortie sonore de son propre patch sans que cela ne vienne affecter celui d'un autre utilisateur connecté et exécutant le même patch. La fréquence de l'oscillateur est ici à considérer comme une constante au traitement. Pour la rendre variable, il suffit, par exemple, à l'un des utilisateurs de remplacer l'objet *sig~* par une *boîte-nombre*, pour ainsi rendre la fréquence variable et paramétrable de façon locale par chaque utilisateur[128].

[128] Notons que Kiwi dispose aussi d'un mécanisme qui permet de programmer une synchronisation des valeurs de jeu entre les participants. Nous exposerons à la sous-section 6.2.1 (p. 219) une autre version du patch présenté ici.

4.4. Autres composants de l'espace de production local

L'espace de travail commun au sein de l'environnement de patching Kiwi se situe exclusivement au sein de la fenêtre Patcher lorsque celle-ci sert à visualiser un document connecté à un serveur. Les autres fenêtres de l'application restent quant à elles locales, c'est-à-dire que les données qu'elles contiennent ou l'interface de contrôle qu'elles exposent ne sont pas partagées entre les différents utilisateurs du logiciel. Parmi celles-ci on trouvera la fenêtre de console qui permet l'affichage de messages, une fenêtre nommée *Beacon Dispatcher* qui permet l'envoi de messages aux patchs ou encore la fenêtre de réglages des préférences audio. Nous détaillons le fonctionnement de ces différents composants dans les trois parties suivantes.

4.4.1. Console

La console est une fenêtre de Kiwi qui sert principalement à afficher des messages destinés à informer l'utilisateur durant l'exécution du programme[129]. Les messages postés dans la console de Kiwi sont locaux, chaque utilisateur dispose donc de son propre historique de messages, même si plusieurs utilisateurs sont connectés aux mêmes patchs. C'est l'un des premiers composants graphiques que nous avons développés parallèlement au Patcher. En effet, au début du projet, nous n'avions pas encore effectué la mise en œuvre des objets graphiques de contrôle au sein de l'application Kiwi. Nous n'avions donc aucun moyen de vérifier le comportement des objets au sein de l'application. La solution a donc été de développer un objet *print*, parallèlement à ce composant graphique, pour permettre aux patchs d'émettre des messages qui peuvent être visualisés et stockés à l'extérieur de la fenêtre.

[129] Elle comprend aussi d'autres fonctionnalités comme le fait de faciliter l'ouverture des documents kiwi par une opération de glisser-déposer (fonctionnalité introduite à la version *v1.0.2*).

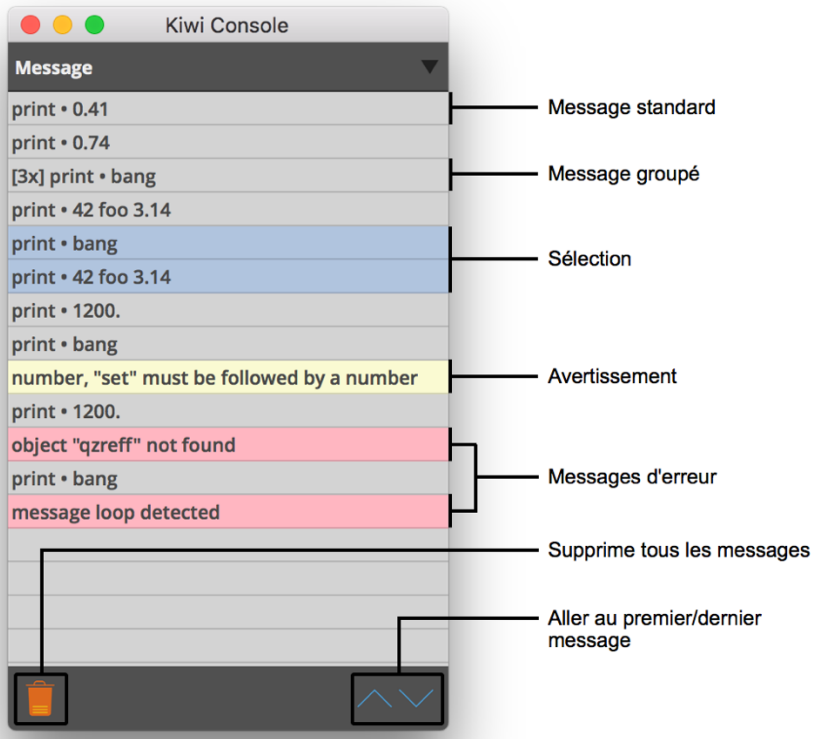

Figure 32 – Capture d'écran exposant la fenêtre de console de Kiwi avec une légende expliquant le type des messages postés et le rôle des différents composants graphiques qui la compose.

Chaque ligne correspond à un message spécifique ; les plus récents étant ajoutés en dessous des plus anciens. Ce composant graphique intègre en bas une barre d'options contenant des boutons qui permettent d'effacer le contenu de la console, de remonter au premier ou d'aller au dernier élément posté [Figure 32]. Chaque message sélectionné peut aussi être copié au sein du presse-papier global du système pour être collé dans une autre application en vue d'être sauvegardé ou partagé si nécessaire, cette fonctionnalité vise en ce sens à faciliter la communication des erreurs locales entre les participants.

Il était important de faire en sorte, lors de la conception de ce système de notification, que l'utilisateur puisse aussi facilement différencier les messages en fonction de leur niveau

d'importance. C'est pourquoi nous distinguons trois types de messages pouvant être postés en console : les messages de type informatif ou de débogage, d'avertissement et d'erreur.

Le premier type de messages est le plus commun, affichés sans couleur de fond particulière, ceux-ci peuvent être postés par l'application elle-même, mais sont la plupart du temps produits par les objets *print* contenus dans les patchs. Cet objet, qui sert d'interface pour la console, peut par exemple être utilisé de manière temporaire pour déboguer un patch en envoyant à la console le résultat d'une opération afin d'en vérifier l'exactitude, ou encore pour tracer le fait qu'une action particulière s'est déroulée. Notons que l'objet *print* peut comporter un argument optionnel qui viendra remplacer le préfixe « print » d'une ligne de messages standard apparaissant dans la console.

Le second type de messages, affiché avec une couleur de fond jaune, représente un simple avertissement. Ce genre de messages est utilisé pour signifier à l'utilisateur qu'une erreur d'importance mineure s'est produite dans l'exécution du programme. Dans l'illustration ci-dessus, le message d'avertissement nous informe d'une mauvaise utilisation de l'objet *number* (boîte-nombre) en indiquant que celui-ci n'accepte le message *set* que s'il est suivi d'un nombre, l'utilisateur peut alors modifier son patch en conséquence pour que ce message n'apparaisse plus et que le patch fonctionne de nouveau normalement.

Le dernier type de messages, affiché avec une couleur de fond rouge, représente une erreur critique. Plus important qu'un avertissement, ce message témoigne d'un problème survenu au cours de l'exécution de l'application (problème lié au réseau par exemple quand l'application ne parvient pas à se connecter au serveur) ou au sein d'un patch. Le premier message d'erreur représenté dans la figure ci-dessus est généré quand l'utilisateur tente de créer un objet qui

n’existe pas ou qui ne peut pas être créé car les arguments passés sont incompatibles ; le second message est affiché lorsqu’une boucle est détectée dans le flux de messages d’un patch, appelé aussi débordement de la pile d'exécution (*stack-overflow*). L’utilisateur est alors invité dans ces deux cas à faire le nécessaire pour que ces messages n’apparaissent plus (modifier le nom de l’objet tapé ou supprimer un lien pour empêcher une boucle).

A la différence des messages standard, les messages d’avertissement et d’erreur ne peuvent être produits que par Kiwi, il n’existe pour l’instant pas d’objet servant d’interface pour permettre à l’utilisateur de les générer à partir d’un patch comme on pourrait le faire avec le logiciel Max grâce à l’objet *error*. Néanmoins, la mise en œuvre d’un tel objet ne pose pas de difficulté particulière et il sera sans doute rajouté dans une future version de l’application.

La console de Kiwi est très proche dans son apparence et son fonctionnement à celles que l’on trouve dans Max et Pure Data mais quelques différences sont néanmoins à noter. Dans Pure Data il est possible de filtrer les messages en fonction d’un niveau de log spécifique parmi les cinq proposés (*fatal* / *error* / *normal* / *debug* / *All*), dans Max il est possible de n’afficher que les erreurs. Même s’il a été envisagé, Kiwi ne propose pour l’instant aucun système de filtrage de ce type dans sa console. Une autre fonctionnalité que nous aimerions ajouter, et qui existe déjà à la fois dans Max[130] et dans Pure Data[131], est la possibilité de retrouver l’objet expéditeur d’un message spécifique en console. Cette fonctionnalité est très utile notamment pour déboguer un patch contenant des erreurs. La dernière différence que nous soulignerons ici est que contrairement au même composant dans les logiciels homologues, dans la console de Kiwi, si un message avec le même contenu est posté plusieurs fois successivement, celui-ci

[130] En double-cliquant sur un message.

[131] En appuyant sur *cmd*/*ctrl*+*click* sur un message

n’apparaîtra qu’une seule fois, groupé et préfixé par le nombre de fois où il a été posté entre crochets ; ce système a été développé pour éviter de surcharger inutilement la console lorsque le même message d’erreur revient plusieurs fois successivement, ou encore pour marquer plus clairement le nombre de répétitions au sein d’un groupe de messages de même origine ou nature. Ce comportement est observable à la troisième ligne de la console présentée à la [Figure 32].

4.4.2. Beacon Dispatcher

Cette fenêtre a été conçue et ajoutée très tôt au sein de Kiwi[132], ici encore, pour s’assurer du bon fonctionnement de l’application dans les premières phases du développement. L’enjeu était de pouvoir interagir avec le patch en transmettant aux objets des messages pour tester leur comportement. Nous avions à cette époque deux solutions. La première était de commencer le développement des objets graphiques – qui rendent effectivement possible cette interaction – cependant les spécifications graphiques n’étaient pas encore clairement définies. La seconde solution, pour laquelle nous avons alors opté, a été de contourner légèrement le problème en adoptant une stratégie alternative et moins coûteuse en temps de développement. Cette approche a consisté à mettre en place un système proche du mécanisme de *send* et de *receive* que l’on trouve aussi dans Max et Pure Data permettant d'envoyer des messages aux objets depuis l’extérieur du patch à travers une interface graphique externe dédiée. Cela nous a permis de commencer à faire les premiers tests de fonctionnement de l’application en attendant de mieux définir les spécifications des objets graphiques de Kiwi.

---

[132] On la trouve dès la version v0.0.2 de Kiwi: https://github.com/Musicoll/Kiwi/releases/tag/v0.0.2 [consulté le 20/09/18].

La fenêtre *Beacon Dispatcher* permet donc de transmettre des messages aux objets contenus dans les patchs ouverts de l'application. Elle tire son nom du noyau logiciel de Kiwi dans lequel il est possible pour les objets, de s'accrocher ou de se lier à ce que nous avons appelé métaphoriquement une bouée ou *Beacon*, par l'intermédiaire d'un *symbole*[133] commun.

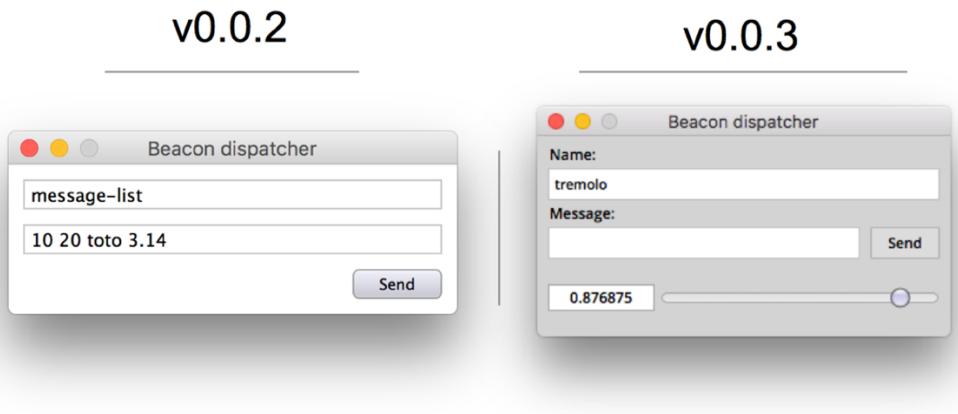

Figure 33 – Capture d'écran de la première version de la fenêtre d'envoi de messages nommée *Beacon Dispatcher* de l'application Kiwi à la version *v0.0.2* (à gauche) et de la version v0.0.3 (à droite). Le champ textuel du haut sert à définir le nom du symbole auquel les destinataires du message doivent être associés (ici le symbole *message-list* et *tremolo*). Le champ textuel du bas sert à spécifier le message à envoyer quand l'utilisateur appuie sur le bouton *send*. Une valeur numérique peut être envoyée en tant que message grâce au *slider* ajouté à la *v0.0.3*, ayant une étendue comprise entre 0. et 1.

Chaque objet lié à un *Beacon* par un symbole spécifique reçoit les messages émis par celui-ci sous la forme d'une liste d'*atoms*. Il est alors possible, à travers l'interface du *Beacon Dispatcher* de transmettre des messages à des objets *receive* spécifiques au sein des patchs, en tapant le symbole auxquels ils se rattachent au sein de champs prévus à cet effet [Figure 33, gauche].

[133] Un symbole est simplement une chaîne de caractères à laquelle des composants logiciels tels que des objets peuvent être liés pour définir une interface de communication fondée sur des messages.

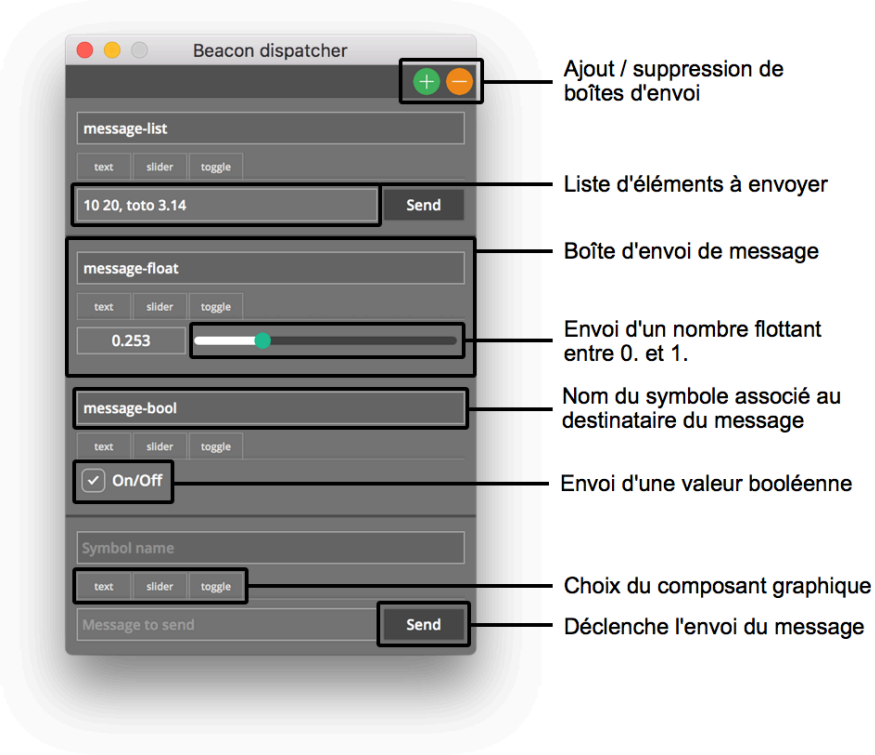

Figure 34 - Capture d'écran de la dernière version de la fenêtre *Beacon Dispatcher* avec une légende expliquant le rôle des différents composants graphiques qui la compose.

La première version de l'interface fonctionnait mais n'était pas suffisamment appropriée pour contrôler un paramètre tel qu'une fréquence ou une amplitude, dans la mesure où nous ne pouvions envoyer que des valeurs discrètes. Un premier raffinement de l'interface a donc été d'ajouter un potentiomètre linéaire (*slider*) permettant d'offrir une variation plus continue des valeurs [Figure 33, droite].

Une dernière version, plus élaboré a par la suite été développée. Celle-ci permet notamment de cibler plus facilement plusieurs destinataires et intègre de nouveaux composants graphiques plus appropriés et spécialisés pour générer des messages simples comme les types booléens (interrupteur) ou des nombres flottants grâce à un *slider* avec une étendue standard entre 0. et 1. [Figure 34].

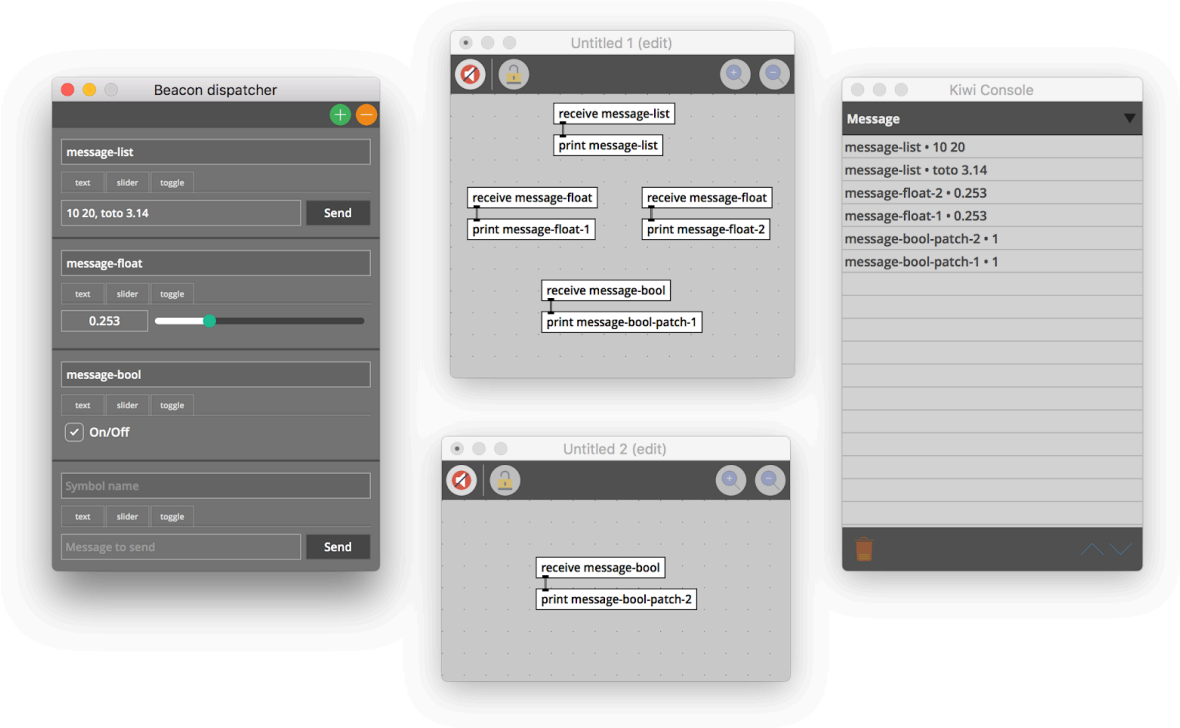

Figure 35 - Capture d'écran de l'interface de l'application Kiwi à la version *v1.0.0*, illustrant l'utilisation du *Beacon Dispatcher*. À gauche la fenêtre nommée *Beacon Dispatcher* contient une liste de quatre composants graphiques permettant d'envoyer des messages spécifiques aux patchs contenant des objets liés aux symboles associés.

La [Figure 35] expose la fenêtre *Beacon Dispatcher* en situation de jeu. Les messages sont envoyés à tous les objets actifs de Kiwi liés à un symbole particulier ; que ce soit au sein du même patch comme pour le patch *Untitled 1* avec le *symbole message-float* associé aux deux objets *receive* ; ou entre différents patchs comme dans le cas des objets *receive* associés au symbole *message-bool* dans les patchs *Untitled 1* et *Untitled 2*. La virgule au sein d'un message textuel agit comme un séparateur. Les éléments entre les virgules sont donc envoyés séparément et de manière séquentielle.

Tout comme la fenêtre de console, la fenêtre *Beacon Dispatcher* ne fait pas partie de l'espace de production mis en commun entre les utilisateurs, l'emplacement de cette fenêtre, les

messages qui y sont tapés et envoyés restent locaux au sein de l'application et ne sont pas partagés avec les autres participants dans un contexte collaboratif.

4.4.3. Préférences audio

Le dernier composant, essentiel à l'interface d'un environnement de patching, et plus généralement à toute application amenée à gérer de l'audio est la fenêtre de réglages des préférences audio. Celle-ci est accessible dans Kiwi depuis le menu général *Options* de l'application et permet de choisir les périphériques d'entrée et de sortie audio, d'activer ou de désactiver certains canaux de sortie et d'entrée et de vérifier que le signal entrant est bien reçu grâce à un bouton de test et un modulomètre de type PPM (*Peak Program Meter*) montrant le niveau de crête du signal entrant[134].

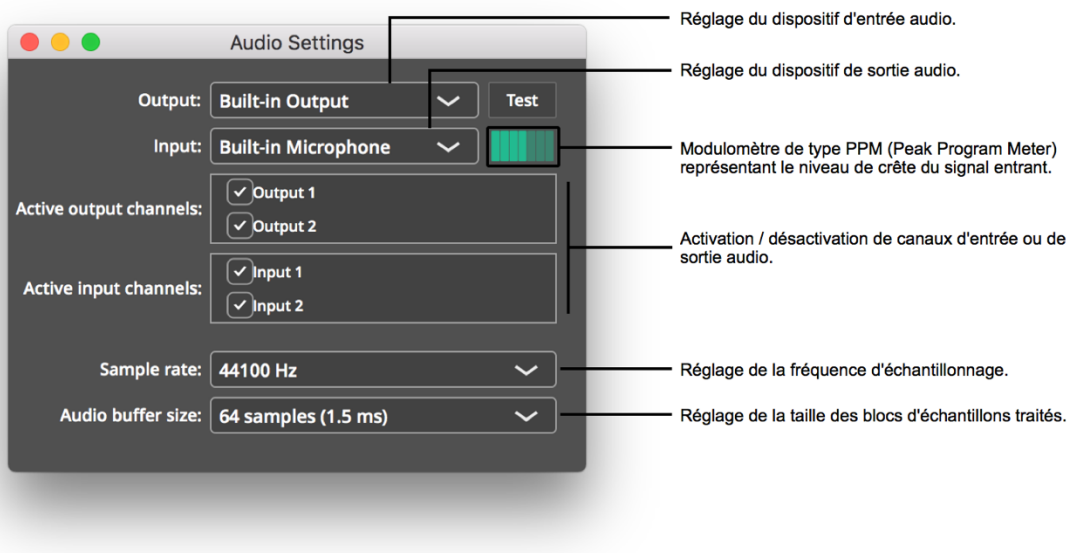

Figure 36 - Capture d'écran de la fenêtre de configuration audio de l'application Kiwi avec légende expliquant le rôle des divers composants graphiques qui la compose.

[134] Cette fenêtre intègre le composant standard de configuration audio disponible au sein de la bibliothèque Juce : https://docs.juce.com/master/classAudioDeviceSelectorComponent.html

Elle offre enfin un accès aux réglages de paramètres audio généraux tels que la fréquence d'échantillonnage ou encore la taille des blocs d'échantillons traités, et fournit le temps de latence induit en millisecondes [Figure 36]. Kiwi ne supporte pour l'instant pas le format MIDI[135] au sein du patch, mais cette fenêtre est déjà prévue pour accueillir les différents réglages qui y sont relatifs. Notons enfin que la configuration audio choisie est enregistrée dans les préférences système de l'utilisateur à la fermeture de l'application pour pouvoir être restaurée au lancement suivant. D'autre part, on pourra aussi préciser que l'ensemble de ces réglages sont locaux, c'est-à-dire qu'ils ne sont pas synchronisés entre les utilisateurs lorsqu'ils sont connectés aux mêmes patchs.

Nous venons de présenter dans ce chapitre les aspects de conception relatifs à l'espace de production du logiciel Kiwi qui peut être utilisé, comme nous l'avons vu, dans un contexte local ou collaboratif. Lorsqu'un patch est utilisé dans un contexte collaboratif, l'espace de travail du patch est mis en commun entre les utilisateurs connectés à une même session et ils partagent alors les mêmes informations issues d'une réplique du modèle de données. Dans ce contexte spécifique les utilisateurs ont besoin de solutions qui vont les aider à coordonner leurs actions.

5. Conception de l'espace de coordination

L'espace de coordination au sein d'un logiciel multi-utilisateur, tel que défini par C. A. Ellis et D. Salber [Ellis & Wainer, 1994], [Salber, 1995], [Figure 3, p. 31], comprend des fonctions qui permettent aux utilisateurs de coordonner leurs actions à travers le collecticiel. Cette coordination s'effectue à plusieurs niveaux. Au sein de l'espace de production du patch, elle vise principalement à l'amélioration des interactions entre les personnes en assistant

[135] Musical Instrument Digital Interface

l'orchestration des actions de groupe. Ce point sera abordé dans la prochaine partie. Un autre aspect de la coordination se situe aussi dans la gestion des sessions, c'est-à-dire dans la manière de gérer l'accès aux différents espaces de travail virtuels que représentent les patchs accessibles en ligne. Nous aborderons ce point dans un second temps. Nous aurons alors l'occasion d'apporter quelques précisions sur le type d'architecture réseau mis en place au sein de Kiwi et sur son fonctionnement.

5.1. Coordination de l'espace de travail partagé

Nous avons formalisé dans la première partie de ce document un certain nombre d'enjeux liés à la coordination de l'espace de travail partagé et notamment le fait de devoir fournir des mécanismes de conscience de groupe permettant aux utilisateurs de se coordonner en leur fournissant des informations sur l'activité des autres utilisateurs au sein d'un environnement graphique virtuel où les utilisateurs ne peuvent plus recourir à des interactions directes pour le faire[136]. Nous étudierons les solutions qu'apporte l'interface graphique du composant Patcher de Kiwi pour répondre à ces enjeux et permettre notamment une conscience de la présence des autres utilisateurs au sein de l'espace partagé (5.1.1), de leur activité d'édition à travers la visualisation de leurs sélections (5.1.2), la mise en place d'animations (5.1.6) ou encore en leur fournissant un contexte d'édition local dans lequel effectuer leurs actions (5.1.3) et (5.1.4). Un autre enjeu fondamental à la coordination des actions entre les participants à une session collaborative est la manière de coordonner l'activité d'édition pour leur permettre de réaliser les opérations qu'ils souhaitent au sein de l'espace de production de manière concurrentielle, sans que cela ne génère de conflits. Cet enjeu, qui est celui du contrôle de la concurrence, sera abordé plus spécifiquement dans la sous-section (5.1.5).

[136] Voir section 3.6 (p. 96) de ce document.

5.1.1. Conscience de connexion et de présence des utilisateurs au sein de l'espace partagé

À mesure qu'une ou plusieurs personnes rejoignent ou quittent le document commun qu'est le patch, les autres participants doivent être en mesure de prendre conscience de leur arrivée et leur départ, un peu comme si ces personnes entraient ou sortaient physiquement d'une pièce. D'autre part, comme le logiciel est ouvert, et que tout le monde peut rejoindre les différents patchs[137], on ne sait pas forcément avec qui on collabore, il peut donc être utile de fournir quelques informations à l'ensemble des participants afin qu'ils sachent avec qui ils travaillent. L'interface du patch change dans Kiwi en fonction du contexte de chargement et d'édition du document. Nous avons décrit ses principales fonctionnalités liées à l'édition dans le précédent chapitre et ses spécificités lorsque le patch est chargé depuis un fichier local [Figure 25, p. 142].

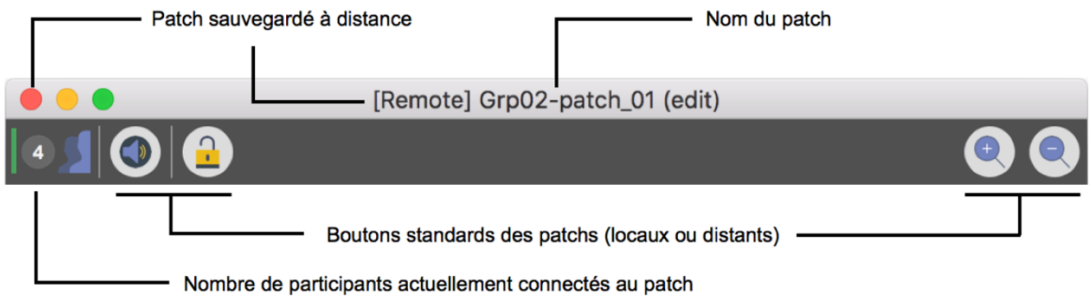

Figure 37 – Capture d'écran présentant la barre de titre et la barre d'options d'un patch chargé depuis le serveur avec une légende expliquant le rôle des différents boutons et composants graphiques qui la compose.

Dans le cadre d'un document chargé à distance depuis un serveur, l'apparence et le comportement de la barre de titre ainsi que de la fenêtre s'adaptent légèrement pour notifier l'utilisateur de l'état de connexion [Figure 37]. Le titre du patch sur la fenêtre est précédé du

[137] Comme nous pourrons le voir dans la partie dédiée à la gestion des sessions au sein de Kiwi (section 5.2, p. 190), la mise en œuvre actuelle de l'application donne un accès public à tous les documents présents sur le serveur.

texte « [Remote] » pour le différencier d'un patch local, de la même manière, le bouton natif n'indique plus que le patch comporte des modifications non sauvegardées, et la commande de sauvegarde est d'ailleurs désactivée puisque le patch est dans ce cas sauvé automatiquement sur le serveur[138].

Une autre différence notable au sein de l'interface graphique, cette fois-ci sur la barre d'outils du patch, est l'apparition de l'icône sur la gauche qui indique le nombre de participants actuellement connectés au document. Une légère animation est déclenchée et le compteur est incrémenté ou décrémenté en fonction des notifications de connexion/déconnexion des utilisateurs au sein du patch en temps réel.

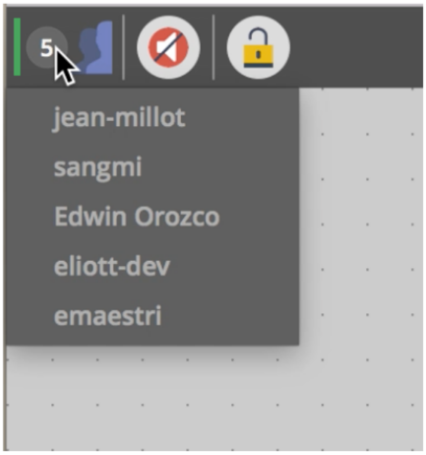

Figure 38 – Capture d'écran de la partie supérieure-gauche d'un patch Kiwi édité à distance exposant une liste des utilisateurs connectés au patch courant dans une fenêtre temporaire.

D'autre part, ce composant graphique expose aussi, sous forme de liste au sein d'une fenêtre temporaire rendu visible par un clic sur l'icône, le nom de l'ensemble des utilisateurs actuellement présents sur le document [Figure 38]. Cette liste est dynamique et se met à jour

[138] Notons qu'il est toujours possible d'obtenir une version locale d'un patch distant en le téléchargeant directement depuis l'interface dédiée à la gestion des documents distants (5.2.2.3, p. 206).

automatiquement en fonction des personnes qui rejoignent ou quittent le patch. Le texte présentant le nom des participants est pour l'instant statique, mais on pourrait imaginer par la suite qu'il puisse réagir au clic ou au survol de la souris pour fournir des informations plus générales et fournies sur les utilisateurs comme un profil plus détaillé, ou encore servir de passerelle pour révéler son activité ou sa localisation précise au sein du patch.

5.1.2. Conscience de l'activité d'édition du patch par le groupe

La sélection des objets et des liens a une double fonction dans Kiwi. Elle sert, tout comme dans un logiciel de patching traditionnel mono-utilisateur, à effectuer des opérations sur les objets (*e.g.* édition, déplacements, redimensionnement des objets ou encore suppression des liens ou des objets). Mais elle peut aussi revêtir une autre fonction dans un contexte collaboratif, celle d'informer les participants connectés de la « place » d'un utilisateur au sein du patch, donner un indice sur la zone du patch où son attention se porte, une information sur l'action qu'il va réaliser ensuite, en d'autres termes offrir une première solution ergonomique à la problématique de la conscience de groupe. Dans un contexte pédagogique, les sélections peuvent aussi aider un utilisateur à pointer simplement un objet ou un groupe d'objets au sein d'un patch pour pouvoir en expliquer le fonctionnement aux autres utilisateurs présents avec lesquels il peut discuter en face-à-face, ou, dans un contexte où les participants se trouvent distribués géographiquement, comme lors d'un rendez-vous virtuel assisté par un logiciel d'audioconférence ou de visioconférence, de pouvoir se référer à un élément commun au sein de l'espace partagé.

Dans la majorité des logiciels d'édition collaborative, des systèmes graphiques permettant aux utilisateurs de voir la sélection des autres participants ont été mis en place pour les mêmes raisons. La manière de le faire en revanche n'est pas standardisée et dépend bien souvent du

domaine d'application et de l'espace de production spécifique auquel s'applique la collaboration.

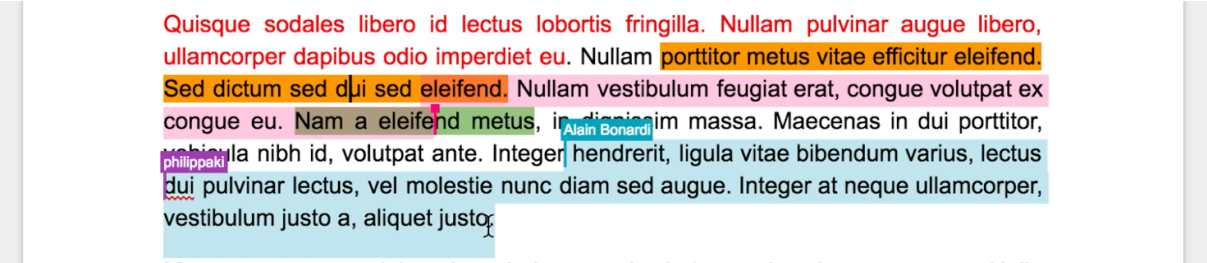

Figure 39 – Capture d'écran d'une partie de document textuel *Google Docs* sur lequel quatre personnes sont connectées et travaillent en temps réel. Le curseur des utilisateurs ainsi que l'état de leur sélection respective sont représentés par différentes couleurs par surbrillance au-dessus du texte. La sélection de l'utilisateur local n'est ici pas représentée.

Dans des solutions professionnelles d'édition collaborative fondées sur du texte telles que *Google Docs*, ou encore *EtherPad* [139], l'application attribue généralement une couleur spécifique à chaque utilisateur qui est ensuite utilisée pour représenter leur curseur et leur sélection au sein du document [Figure 39]. Les curseurs présents sur le document peuvent aussi indiquer le nom de l'utilisateur à qui il appartient. Ce système paraît tout à fait pertinent dans la mesure où il permet bien de distinguer chaque utilisateur séparément au sein de l'espace de production partagé, et l'endroit précis où ils sont en train de travailler. Nous sommes donc ici en présence d'un système qui gère à la fois la conscience de présence [Greenberg, 2002] et de localisation des autres utilisateurs au sein de l'espace partagé.

Néanmoins, cette solution graphique présente selon nous aussi des inconvénients qu'il nous faut exprimer afin de mieux comprendre les choix qui ont été faits dans le cadre de la mise en œuvre d'un système similaire au sein de l'interface graphique de Kiwi. Dans ce type de système

[139] http://etherpad.org/.

il faut pouvoir disposer d'une couleur par utilisateur au sein de l'interface. Or, il n'y en a généralement pas assez – qui contrastent suffisamment les unes avec les autres – pour en attribuer une de manière fixe à chaque utilisateur. Le même utilisateur peut donc être représenté dans une certaine couleur chez quelqu'un d'autre, puis dans une couleur différente à un autre moment ou au sein d'un autre document, ce qui peut se révéler au final assez perturbant. Nous pouvons en fait analyser ce phénomène comme un excès de relâchement du principe de *WYSIWIS*[140] dans la mesure où les utilisateurs ne peuvent plus, au sein d'un groupe hétérogène de participants, se référer à une couleur pour parler de la contribution ou de l'action spécifique de l'un d'entre eux. Rien ne dît par exemple qu'une injonction telle que « *regardez la partie sélectionnée en violet* » puisse encore signifier quelque chose ou être pertinente pour tous les participants lorsqu'elle est prononcée dans le contexte d'une session multi-utilisateur. D'autre part, si le choix des couleurs multiples peut se révéler pertinent dans le cadre d'un espace de production faisant intervenir uniquement des opérations sur du texte, dans lequel un curseur discret suffit à notifier la présence d'un utilisateur, et où les couleurs dominantes sont principalement celles d'un texte écrit en noir sur blanc, il l'est beaucoup moins dans une application plus importante qui utilise une palette de couleurs plus large pour d'autres éléments de son interface. Cette dernière a alors vite tendance à se transformer en « arc-en-ciel », et ne permet plus de distinguer aussi nettement ce qui relève de la sélection de ce qui relève des autres éléments utiles de l'espace de production. Ce phénomène est d'ailleurs aussi observable au sein de l'espace de travail partagé de *Google Docs* dès que d'autres couleurs que le noir sont utilisées au sein de l'interface, la couleur des sélections se confondant avec la couleur de fond du texte ou des éléments surlignés [Figure 39].

[140] Nous nous référons ici au principe décrit précédemment dans la section 3.7 (p. 99) de ce document.

Nous avons donc choisi de résoudre autrement cette problématique au sein de Kiwi en ne se servant que d'une seule couleur pour notifier de la sélection des différents éléments par les autres utilisateurs au sein du patch. Ce choix ergonomique, décrit notamment dans [Paris & al., 2017], nous a été suggéré par l'équipe *OhmForce* lors de l'une de nos réunions de travail dans le cadre du projet MUSICOLL. Dans le cas du séquenceur multi-utilisateur *OhmStudio*, le choix d'une couleur de sélection unique a aussi été fait dans la mesure où d'autres couleurs servaient déjà à signifier autre chose au sein de l'interface (les utilisateurs utilisent par exemple généralement une couleur par piste ou par instrument au sein d'une session multipiste). Ajouter une couleur différente pour chaque utilisateur aurait amené dans leur cas à une surcharge d'informations inutile au sein de l'interface graphique entamant alors d'autant sa lisibilité. Dans le cas des langages de programmation visuelle traditionnels, tels qu'ils sont mis en œuvre dans Max ou Pure Data, les objets comportent souvent plusieurs couleurs (*e.g.* les différentes *leds* d'un objet *meter~*), d'autre part, ces couleurs peuvent aussi être configurées au sein de l'interface directement par l'utilisateur afin de mieux organiser son patch ou permettre une meilleure lisibilité ou utilisabilité[141]. Le choix d'employer une couleur unique s'est donc imposé aussi dans Kiwi pour ces raisons.

[141] Un exemple a été donné dans ce document à la [Figure 5, p. 44]. Dans ce contexte les utilisateurs du logiciel Max usaient de la configuration des couleurs des objets pour créer un système permettant de mettre en lumière la contribution de chacun au sein du patch édité de manière collaborative. Cette fonctionnalité n'est néanmoins pas encore disponible au sein de Kiwi.

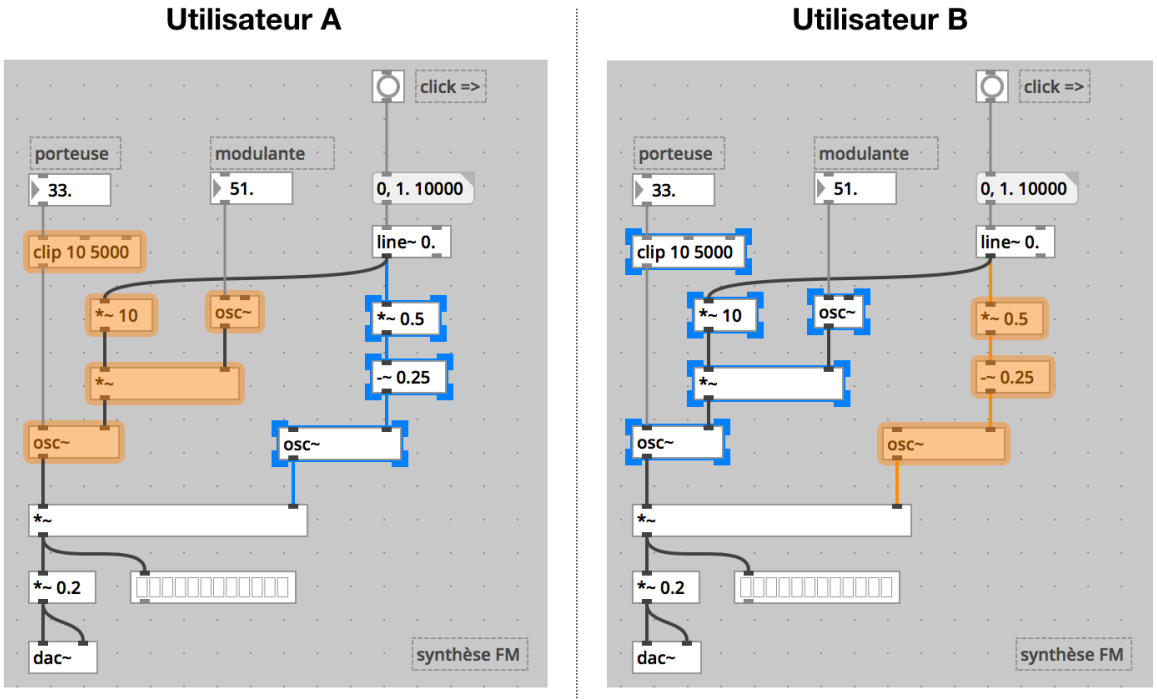

Figure 40 – Capture d'écran du même patch Kiwi chez deux utilisateurs différents connectés au même document. La sélection locale des éléments est marquée chez chacun d'eux par un rectangle bleu, les éléments sélectionnés par l'autre utilisateur au sein du patch sont représentés en orange.

Lorsque plusieurs utilisateurs sont connectés à un document, les éléments qu'ils sélectionnent eux-mêmes sont, comme dans le cadre d'une utilisation hors-ligne du patch, représentés en bleu. En revanche, ils ont aussi la possibilité de voir ce que les autres utilisateurs sélectionnent (objets et liens) en temps réel sur l'espace partagé du patch.

Les sélections des autres utilisateurs s'affichent alors chez eux en orange [Figure 40]. Il n'est pas possible de savoir précisément qui sélectionne quel objet au sein du patch[142]. Ce système permet alors simplement aux utilisateurs de s'apercevoir qu'un (ou plusieurs) utilisateur(s) est ou sont en train d'effectuer une action sur un certain groupe d'objets ou de liens au sein du patch. Ce mécanisme de notification de l'activité des autres sur le patch par l'intermédiaire des

[142] Notons que même si on utilise qu'une seule couleur de sélection, rien ne nous empêche d'afficher une information plus granulaire et détaillée, ailleurs au sein de l'interface, pour décrire plus précisément qui sélectionne un élément au sein du patch ou réalise telle ou telle action. Néanmoins ces informations, présentes dans le *OhmStudio*, sont pour l'instant absente dans Kiwi.

sélections est donc assez simple. Il est néanmoins complexifié dans le cadre de Kiwi par plusieurs éléments.

Comme nous l'avons vu précédemment, un utilisateur peut avoir plusieurs vues du même document dans Kiwi, et les sélections locales peuvent être différentes sur chacune de ces vues, cf. [Figure 30, p. 149]. Nous aurions alors pu colorer simplement la sélection de la vue courante et ignorer le fait qu'un objet soit sélectionné dans une autre vue (comme c'est le cas dans Max par exemple). Il nous a néanmoins paru pertinent d'effectuer une distinction entre la sélection sur la vue courante et la sélection d'un même élément sur une autre vue ; ceci pour couvrir des cas où un utilisateur disposerait d'une vue sur son ordinateur et d'une autre qu'il projetterait sur un écran – utile par exemple dans le cadre d'un cours où un professeur veut exposer une autre vue de son patch aux élèves en en gardant une sur sa machine. Ce dernier peut alors se servir de la sélection pour pointer une partie du patch sur sa vue locale et faire apparaître cette information sur les autres vues. Des systèmes tels que *Google Docs* font d'ailleurs aussi cette distinction. Un même utilisateur peut charger plusieurs fois un même document sur une ou plusieurs machines, le curseur ainsi que l'étendue de sa sélection apparaît alors sur chacune des autres vues dans une couleur, en l'occurrence verte, contrastant avec la sélection locale de la vue courante.

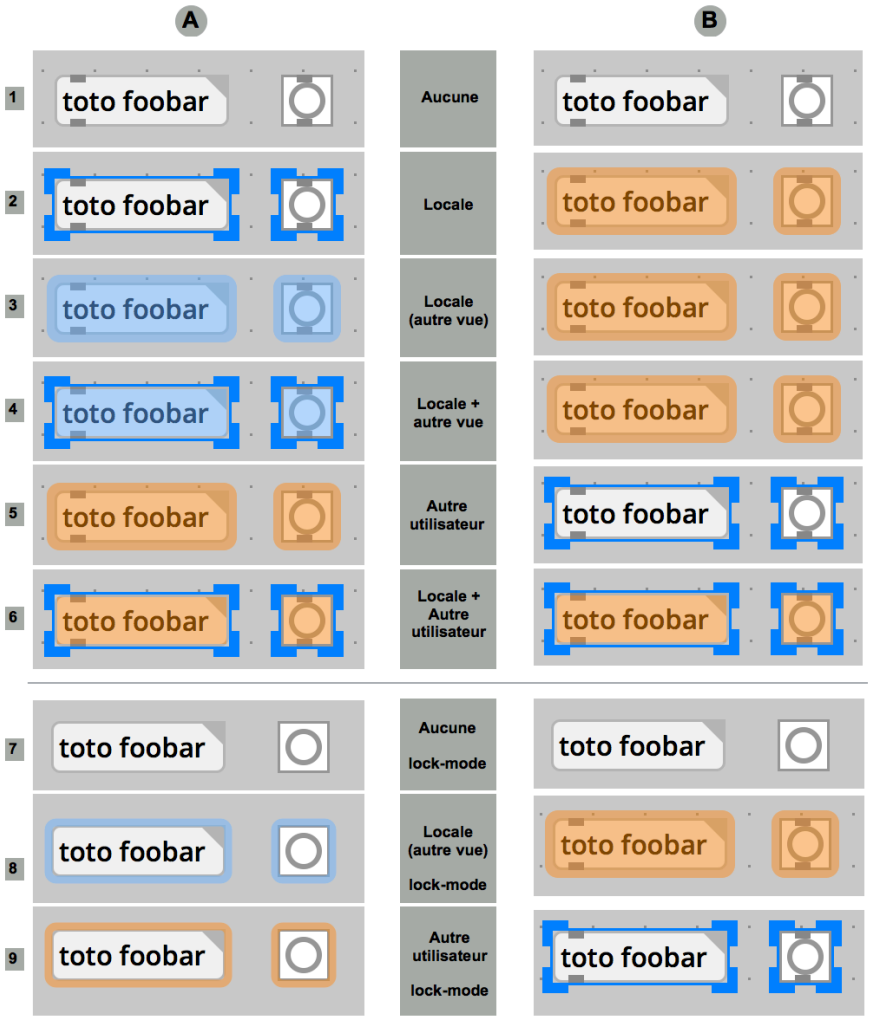

Figure 41 – Tableau présentant les différents modes de coloration des sélections d'objets dans Kiwi (depuis la *v1.0.2* du logiciel), en fonction de la sélection de la vue courante, de la sélection locale sur les autres vues du même patch, de la sélection des autres utilisateurs au sein du patch et enfin du mode de jeu. Les lignes 1 à 6 correspondent à la vue d'un patch en mode édition, les lignes 7 à 9 en mode *lock* ou jeu. Les informations de sélection sont données par rapport à la première vue de l'utilisateur A, situé à gauche qui dispose de plusieurs vues du même patch. La partie de droite représente l'état de l'objet tel qu'il est visualisé chez l'utilisateur B, en fonction du type de sélection.

D'autre part, les sélections peuvent aussi être gênantes, notamment lorsque l'on est en mode *jeu* au sein du patch, si elles couvrent l'ensemble d'une interface de contrôle. Il a donc été décidé dans la dernière refonte de l'interface graphique de Kiwi de colorer différemment les sélections en fonction du mode courant de jeu de chaque utilisateur. Nous avons donc dans Kiwi une interface dans laquelle le mode de sélection, à la base binaire et très simple des

éléments au sein de l'espace de production (soit local, soit distant) est complexifié par ces différents cas spécifiques et est conditionné par plusieurs éléments (vues différentes, mode d'interaction courant du patch, sélections locales ou distantes). Ces différents modes d'affichage des sélections sont explicités par les lignes et les deux figures qui suivent.

La figure ci-dessus présente de manière exhaustive l'ensemble des combinaisons d'état de sélection possibles au niveau des objets au sein d'un patch [Figure 41].

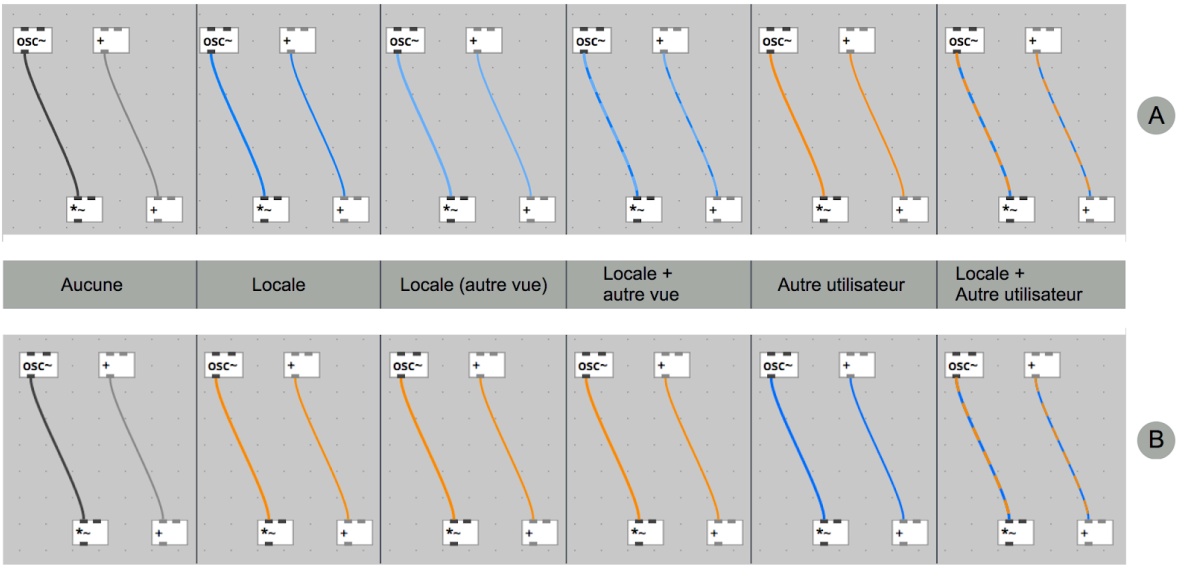

Figure 42 – Capture d'un patch Kiwi à la version v1.0.2 montrant de façon exhaustive les différentes manières dont peuvent être colorés les liens au sein du patch pour représenter la sélection courante de l'utilisateur et de celle des autres personnes connectées. La légende est donnée par rapport à la première vue du patch de l'utilisateur A, situé en haut, qui possède plusieurs vues de ce même patch auquel est aussi connecté un utilisateur B, représenté en dessous. Chaque vignette verticale présente un lien de type signal et lien de type message.

La figure ci-dessus présente de manière exhaustive l'ensemble des combinaisons d'état de sélection possibles au niveau des liens au sein d'un patch [Figure 42].

Dans les deux figures présentées ci-dessus, [Figure 41] et [Figure 42], les informations de sélection sont données par rapport au point de vue de chaque utilisateur en fonction de sa sélection (vue courante et autre vue locale) et de celles des autres utilisateurs connectés au

même patch. L'utilisateur A dispose de plusieurs vues locales du même patch. L'apparence des sélections diffère si l'élément a été sélectionné localement sur la vue courante, sur une autre vue ou sur les deux à la fois (utilisateur A, colonne ou ligne 2, 3 et 4), en revanche l'utilisateur B ne distingue pas sur quelle vue ont été sélectionnés les éléments de l'utilisateur A. Les objets et les liens gardent chez lui le même aspect et lui notifient seulement que l'objet a été sélectionné par quelqu'un d'autre (utilisateur B, colonne ou ligne 2, 3 et 4). En effet nous n'avons pas trouvé pertinent de traduire graphiquement cette information ou de faire cette distinction du point de vue des autres utilisateurs. Les seules combinaisons qui génèrent le même rendu graphique chez tous les utilisateurs sont quand un lien ou un objet n'est sélectionné par personne ou qu'il est sélectionné par plusieurs personnes à la fois (colonne ou ligne 1 et 6). Notons aussi que contrairement aux objets, aucune distinction graphique n'est faite entre les deux modes de jeu du patch au niveau de la sélection des liens. Si un élément n'est pas sélectionné sur la vue courante mais qu'il l'est à la fois sur une autre vue locale et par un utilisateur distant, celui-ci s'affichera avec la couleur d'une sélection distante car nous considérons que l'information permettant de savoir si un utilisateur distant sélectionne ou-non un élément prime sur le fait de savoir si un élément est sélectionné localement.

Les sélections représentent un exemple parfait d'adaptation de la vue graphique qui a dû être mis en place au sein de Kiwi en fonction du contexte de chaque utilisateur pour permettre la conscience de groupe au sein de l'espace de production. Elles illustrent aussi le relâchement nécessaire du principe de *WYSIWIS*. Une autre illustration au relâchement de ce principe peut être trouvée dans Kiwi au niveau du temps d'affichage des informations entre les vues de chaque participant.

5.1.3. Relâchement du principe de *WYSIWIS* sur le temps d'affichage

On distingue deux types de modifications pouvant être apportées au document dans Kiwi. Les modifications que l'on pourrait appeler ponctuelles – qui peuvent s'exprimer comme une seule opération, supprimer un objet ou un lien par exemple – et les modifications continues comme l'édition d'un objet, le redimensionnement des objets, leur déplacement, ou encore la création d'un lien, qui peuvent être considérées comme gestuelles.

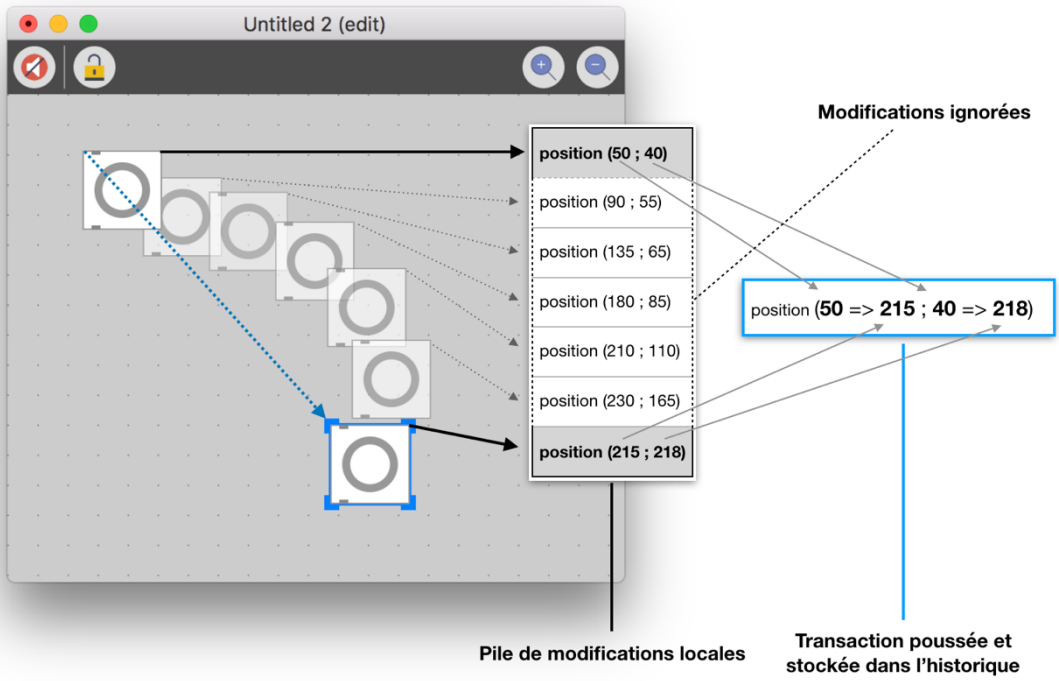

Figure 43 – Représentation du mécanisme de modification de type gestuelle à travers la décomposition d'une action de déplacement d'un objet au sein du patch.

L'idée générale derrière une modification gestuelle est que la modification du document se fait dans le temps, c'est-à-dire que le document est modifié progressivement à mesure que l'utilisateur effectue son action.

Dans l'exemple proposé à la figure ci-dessus, l'utilisateur effectue une opération de déplacement d'un objet au sein du patch [Figure 43]. Cette action n'est pas réalisée en une seule opération mais est constituée d'un certain nombre d'états intermédiaires qui correspondent au

geste effectué par l'utilisateur pour la réaliser. Si l'on décompose cette action, on a un état de départ qui correspond à la position de l'objet lorsque l'utilisateur clique une première fois dessus pour entamer l'action ; puis plusieurs états intermédiaires qui correspondent à la position de l'objet lorsque l'utilisateur le fait glisser à travers le patch ; enfin un dernier état lorsque l'utilisateur relâche la souris, et met fin à cette action.

Les modifications gestuelles sont mises en œuvre sous la forme de ce que nous avons appelé des sessions d'historique au sein du code. Une session d'historique comporte son propre historique de transaction et a pour comportement de ne publier que la dernière opération effectuée (à la fois sur l'historique des transactions locales et aux utilisateurs distants connectés au patch). Ce comportement est nécessaire car il permet notamment au système d'*undo/redo* que nous étudierons dans la sous-section suivante, de proposer à l'utilisateur d'annuler l'opération gestuelle dans son ensemble (ici de déplacement de l'objet) et non-pas chaque opération intermédiaire. Mais il a aussi bien sûr une incidence sur la vue partagée puisque le nouvel état du document n'est partagé aux autres utilisateurs qu'une fois l'action terminée localement. Les autres utilisateurs présents sur le document ne voient donc pas les états intermédiaires des actions de type gestuelles effectuées par les autres utilisateurs. On a donc ici un exemple de relâchement du principe de *WYSIWIS* sur le plan du temps d'affichage tel qu'évoqué dans la première partie de ce document et théorisé par [Stefik & al., 1987].

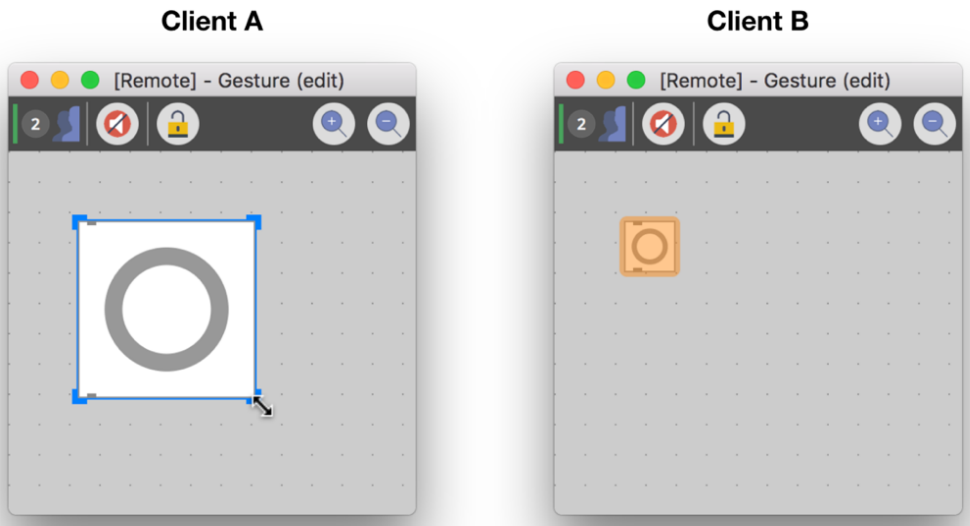

Figure 44 – Capture d'écran représentant le comportement du redimensionnement des objets au sein du patch dans une situation collaborative.

Sur la figure présentée ci-dessus, qui illustre un autre type d'opération gestuelle, la modification de la taille de l'objet exercée par le client A n'est pas encore prise en compte par le client B du fait qu'il n'a pas encore terminé son action [Figure 44]. Le client B, voyant que l'objet est sélectionné, peut seulement en déduire qu'il va potentiellement être modifié bientôt. Cela l'aide donc à coordonner ses action en l'invitant à ne pas toucher à cet objet en même temps pour ne pas venir interférer avec l'action qu'est en train de réaliser l'utilisateur sélectionnant l'objet.

Le fait de mettre en œuvre certaines actions sous la forme de geste ou d'action isolée provient d'un choix arbitraire de notre part ; les sélections au lasso sont actuellement transmises instantanément à tous les utilisateurs, mais nous aurions très bien pu faire le choix de ne partager l'information qu'au moment où l'utilisateur relâche la souris. Dans le cas spécifique des sélections, nous avons pris ce parti dans la mesure où nous avons jugé qu'il était pertinent de partager au plus vite cette information afin de pouvoir faciliter la communication inter-

utilisateur au sein du patch et améliorer ainsi la coordination de leurs actions en leur permettant de prendre conscience de l'endroit où se trouvent les autres utilisateurs grâce à celles-ci.

5.1.4. Contexte local d'annulation et de restauration des actions

L'opération d'annulation et de restauration des actions, plus connue sous le nom d'*undo/redo*, est une commande classique présente dans pratiquement tout logiciel d'édition actuel, qu'il soit graphique ou textuel. À ce titre, cette commande est aussi disponible dans l'application Kiwi, notamment à partir de la barre de menu générale qui expose une description textuelle de la dernière action annulable ou à restaurer. Lorsqu'un utilisateur a fait une modification (*e.g.* ajout ou suppression d'un lien ou d'un objet, ou encore déplacement ou redimensionnement des objets) mais considère que celle-ci n'est pas utile, il peut alors l'annuler en activant la commande d'*undo* qui va remettre le document dans l'état dans lequel il se trouvait avant d'effectuer l'action ; de la même façon, l'opération d'*undo* est d'une certaine manière elle aussi annulable, dans ce cas, en activant la commande de *redo* on va chercher à retrouver l'état dans lequel était le document avant d'avoir annulé la dernière action. Mais dans un contexte collaboratif, comment est censé fonctionner ce système ? L'édition du document se fait à travers des opérations successives, or ces actions peuvent être réalisées par plusieurs personnes différentes. Quand un utilisateur déclenche la commande d'annulation de la dernière action, cherche-t-il à annuler la dernière action réalisée sur le document, ou la dernière action qu'il a lui-même effectuée ? Nous avons déjà évoqué ces questions dans le cadre de l'étude des solutions de patching collaboratif dans la première partie de ce document, à l'occasion de l'étude du projet *peerdata*, et avons montré qu'un système gérant ce mécanisme de manière globale n'était pas adapté dans la mesure où il générerait des incompréhensions chez les utilisateurs qui ne s'attendent pas à ce qu'ils fonctionnent comme tel. Nous en avons alors déduit, grâce notamment à [Prakash & Knister, 1992], que le système le plus adapté pour le

gérer était de fournir un contexte d'exécution de ces opérations qui soit spécifique à chaque utilisateur. Dès lors, comment est géré ce cas dans Kiwi ?

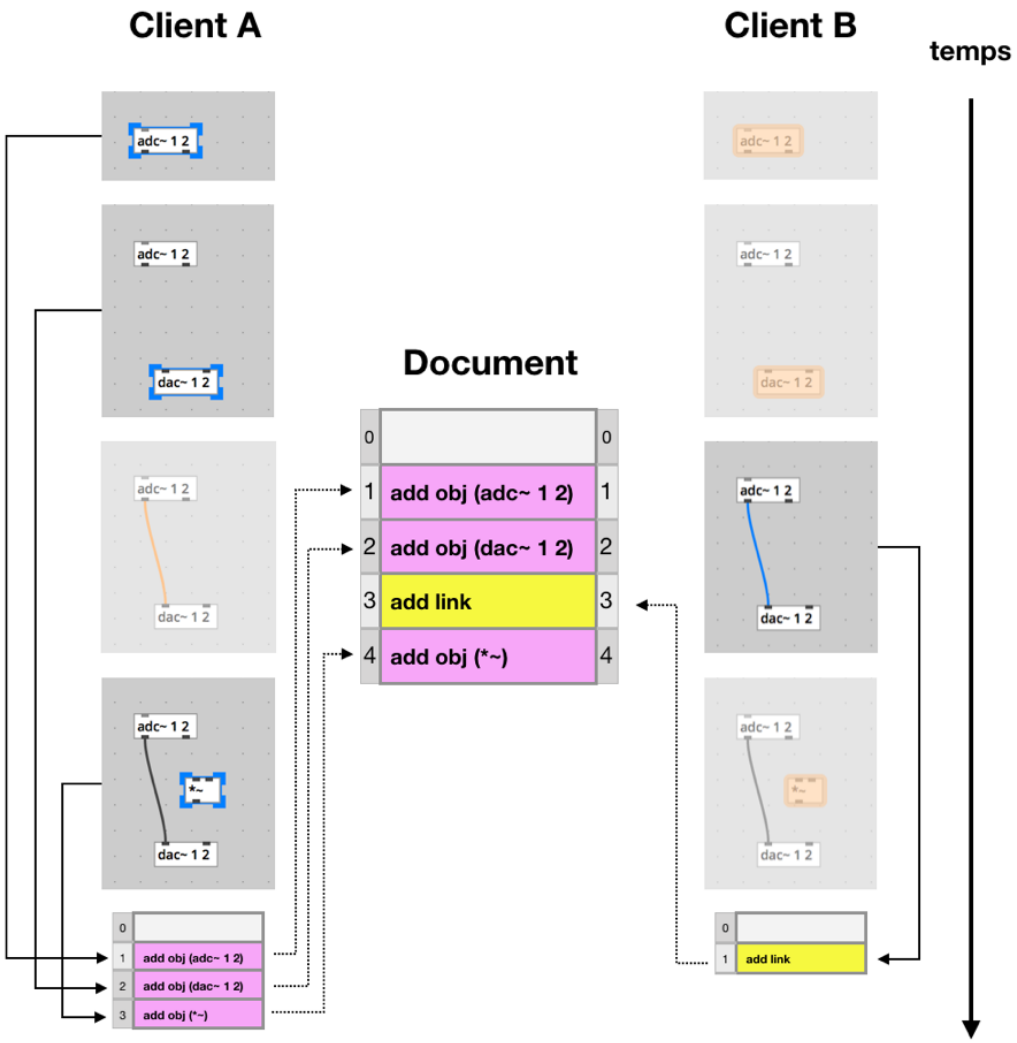

Figure 45 – Schéma présentant de manière simplifiée le mécanisme d'historique de transaction présent dans Kiwi, à travers la modification synchrone d'un document par deux utilisateurs. Les utilisateurs partent tous deux du même document vierge puis réalisent en tout quatre actions (de haut en bas). Le client A en réalise trois tandis que l'utilisateur B n'en réalise qu'une seule.

Le scénario d'édition présenté dans le schéma ci-dessus, [Figure 45], pourrait être le suivant : deux utilisateurs sont connectés à un patch Kiwi qu'ils éditent en même temps. Le client A ajoute deux objets au patch (l'objet *adc~* et l'objet *dac~*). Il vient alors d'effectuer deux opérations successives qui sont appliquées au document local, stockées dans un historique lui-

aussi local, puis poussées vers le document de référence sur le serveur pour être répliquées chez chacun des utilisateurs. Le client B, voulant aider le premier en observant ses actions en temps réel, anticipe alors la prochaine action de son collaborateur et relie la sortie de l'objet *adc~* à l'entrée de l'objet *dac~* (troisième opération globale, stockée en locale dans son historique local et répliquée chez le client B). Voyant que l'action suivante du client A se solde par l'ajout d'un objet multiplicateur du signal « *~ » (quatrième opération globale), l'utilisateur B comprend que l'intention de son partenaire n'était peut-être pas de relier directement l'entrée audio à la sortie mais plutôt d'appliquer un gain sur le signal entrant. Le client B décide alors d'annuler son action en activant la commande d'*undo*. Du point de vue du document, la dernière action réalisée est l'ajout de l'objet multiplicateur de signal, en revanche, celle que l'utilisateur B veut annuler n'est pas celle-ci mais bien la dernière qu'il avait réalisée, c'est-à-dire celle consistant à créer un lien. La bibliothèque *flip* permet une mise en place extrêmement simplifiée du mécanisme d'*undo/redo* adaptée aux enjeux de l'édition multi-utilisateur. Comme nous l'avons dit, chaque modification est stockée, sous la forme d'une transaction, au sein d'un historique local – c'est-à-dire non-pas sur le serveur mais au sein de l'application cliente. Pour un utilisateur, annuler la dernière action revient alors simplement à appliquer au document la dernière transaction stockée localement dans l'historique, dans le sens inverse[143].

5.1.5. Gestion des conflits et validation du modèle de données

Lors d'une édition collaborative synchrone de document, plusieurs utilisateurs peuvent effectuer des modifications de manière simultanée sur les données de celui-ci. Si ces modifications ne sont pas coordonnées au niveau logiciel, elles peuvent alors générer des conflits ou des incohérences venant corrompre le document. Cet enjeu est celui du contrôle de

[143] Ce mécanisme dépend d'une propriété fondamentale des transactions de *flip* qui est de pouvoir être appliquée au document à la fois dans un sens ou dans l'autre.

la concurrence que nous avons exposé à la section 3.5 (p. 94) de ce document. Nous avons alors évoqué deux approches différentes à la mise en place d'un contrôle de la concurrence au sein d'une application multi-utilisateur, à savoir l'approche *pessimiste* qui consiste à bloquer les accès concurrents au modèle de données et l'approche *optimiste* autorisant un accès non-bloquant. Nous avons aussi spécifié, en nous appuyant notamment sur [Greenberg, 2002], qu'un logiciel collaboratif synchrone devait suivre cette dernière approche en gérant les conflits de manière automatique et transparente pour l'utilisateur.

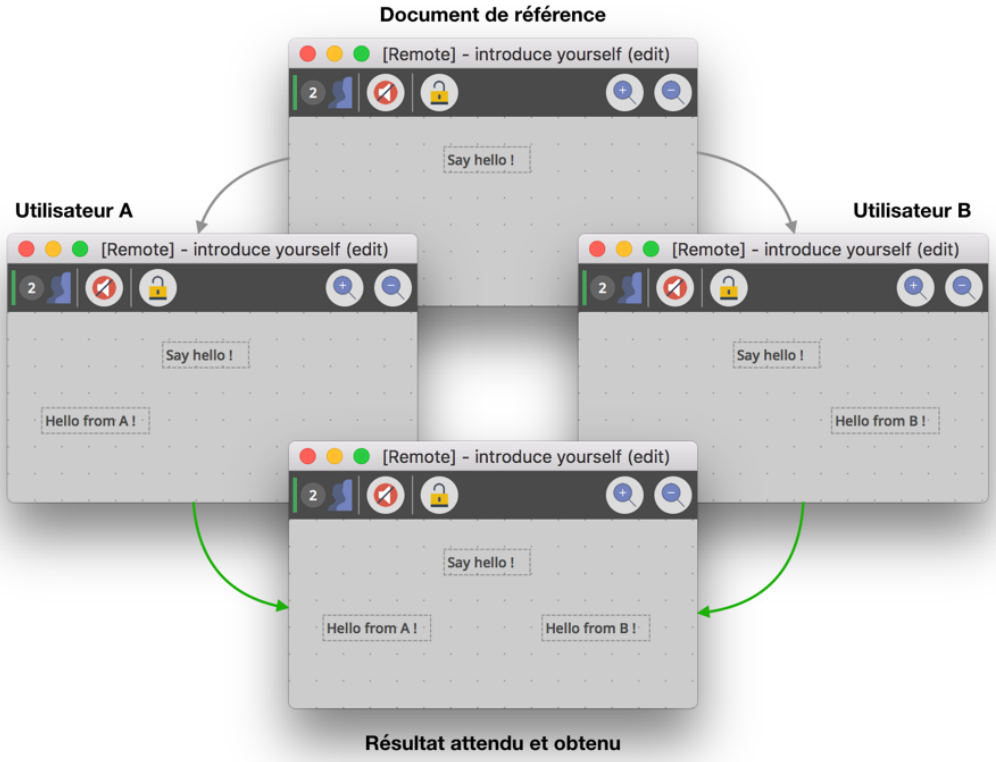

Figure 46 - Schéma représentant une édition parallèle de document réalisée à partir d'un même document de référence par deux utilisateurs et l'état final du document suite à leur modification. L'utilisateur A, à gauche, ajoute un objet *comment* au patch pour se présenter, l'utilisateur B, sur la droite, fait de même. Les deux commentaires se retrouvent fusionnés dans le document de référence sans générer de conflits ni faire appel à une action spécifique de leur part.

La bibliothèque *flip* va dans ce sens en fournissant au programmeur un mécanisme de résolution automatique des conflits. Ce système bénéficie par extension à l'utilisateur en lui offrant la

possibilité d'éditer un document comme s'il était tout seul à y travailler, dans la mesure où les conflits y sont gérés de manière transparente et sans aucune action spécifique requise de sa part.

La [Figure 46] ci-dessus fait écho au cas de conflit rencontré lors de l'étude de la collaboration asynchrone d'un document Pure Data supporté par un logiciel de gestion de version (2.1.2.2, p. 59) et illustré par la [Figure 10, p. 61]. Dans les deux cas les utilisateurs partent d'une version référente de document et y apposent simplement un objet commentaire (*comment*). Dans le cas d'usage de *git*, ces deux opérations se soldaient par un conflit de fusion que l'utilisateur devait alors régler manuellement. Ici tout se passe de manière fluide et on obtient bien le résultat souhaité. La différence fondamentale est qu'ici les deux utilisateurs sont connectés en temps réel au document et donc l'information est propagée beaucoup plus rapidement entre les utilisateurs. Dans la majeure partie des cas, les opérations d'édition se font donc de manière séquentielles et ne génèrent plus de conflits. Mais il peut très bien arriver que les modifications parallèles (transmises sous la forme de transactions) arrivent exactement au même moment sur le serveur, pouvant alors causer de réels problèmes d'édition concurrentielle. L'ajout d'un objet au patch se solde au niveau du code du modèle de données par l'ajout d'un nouvel élément au sein du conteneur d'objets du patch [Figure 21, p. 130, vignette 1, ligne 3], à l'index suivant celui correspondant au dernier élément. Un cas de concurrence se présente donc ici lorsque deux applications clientes tentent d'ajouter un nouvel objet à la même place (au même index) au sein du même conteneur. *Flip* dispose, pour régler ce genre de cas de conflits, d'algorithmes non-bloquants, directement intégrés au système de typage, permettant à plusieurs personnes d'éditer la même structure de données de manière concurrente et au développeur d'effectuer des opérations sur ces types de données comme s'il les effectuait sur des types standard du langage C++. À chaque fois qu'une transaction est appliquée à un document, la technologie *flip* s'assure que la structure du document reste cohérente en exécutant un code de validation. Cette

validation est transparente pour l'utilisateur mais l'est aussi pour le programmeur dans la mesure où la vérification est faite en interne par le système. Par contre, *flip* gère ces conflits de manière générique dans la mesure où il doit s'adapter à la création de tout type d'application. Il permet donc d'insérer ou de supprimer par exemple un élément dans un conteneur de manière concurrentielle, et de valider les cas d'erreurs au niveau de la structure interne du document mais, tout comme le système de gestion de version, il ne connaît bien sûr pas automatiquement le concept de patch, d'objets ou de liens, ni ce que nous considérons comme valide ou non au sein de notre propre domaine d'application. Néanmoins, contrairement au système *git*, *flip* fourni un mécanisme qui permet au programmeur de définir des invariants spécifiques qui permettent de maintenir la cohérence du modèle d'un point de vue sémantique.

Dans la version actuelle de l'application Kiwi nous avons fait le choix de maintenir les invariants suivants au sein de notre modèle de données du patch :

- Un lien doit nécessairement être relié à deux objets valides du patch, c'est-à-dire deux objets qui résident réellement au sein du document lorsque le lien qui les relie est créé.
- Un lien doit être relié à une entrée valide et compatible d'un objet.
- Un lien ne peut pas être relié en entrée et en sortie au même objet.

Certains de ces invariants peuvent être gérés de manière préventive par la vue graphique du patch. Consciente des invariants du modèle, la partie contrôlant de la vue peut être codée de manière à prévenir les incohérences potentielles du document en empêchant certaines actions de se produire.

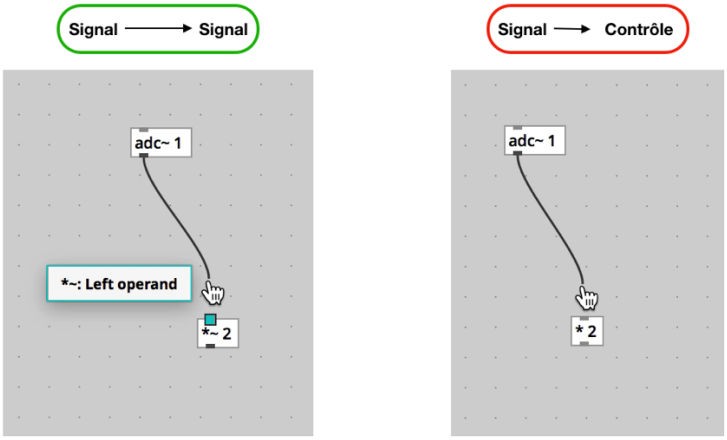

Figure 47 – Capture d'écran exposant le comportement de la vue graphique du patch durant une interaction consistant à créer un lien entre deux objets. Sur la gauche, la connexion que l'utilisateur s'apprête à réaliser est de type signal vers signal, l'entrée de l'objet (*~) lui est alors proposée avec l'apparition d'une fenêtre temporaire. À droite en revanche, aucune proposition de connexion n'est faîte à l'utilisateur dans la mesure où il n'existe pas d'entrée compatible à proximité de la zone pointée par la souris et le lien ne pourra donc pas être créé.

Lorsqu'un utilisateur clique sur une sortie de type signal d'un objet pour créer un lien dans Kiwi, la vue ne propose une connexion qu'avec des entrées acceptant du signal, de façon à prévenir qu'aucun lien de type signal n'arrive dans une entrée ne gérant pas ce type de données [Figure 47]. Cette gestion préventive par la vue est néanmoins assez fragile dans la mesure où rien n'assure que le modèle reste valide si une mise à jour du code lié à la partie vue est effectuée (celle-ci devant normalement interchangeable sans avoir à toucher au modèle dans une architecture de type MVC telle que mise en place dans Kiwi). Le modèle doit donc assurer lui-même la validation de ses données à chacune de ses modifications.

La gestion des conflits au sein d'un patch dans Kiwi passe donc essentiellement par la validation de son modèle de données par l'intermédiaire de la bibliothèque *flip*. À chaque modification publiée du document, une validation est exercée pour s'assurer du maintien de sa cohérence.

Une première validation se fait au niveau du modèle local. Lorsqu'un lien est ajouté au modèle, ce dernier s'assure qu'il soit valide (qu'il respecte les invariants que nous avons fixés) et ne l'ajoute que s'il l'est. De la même façon, lorsqu'un objet est supprimé d'un patch, le comportement mis en œuvre au sein de Kiwi consiste à supprimer aussi les liens qui y sont rattachés (en entrée et en sortie) avant de pousser la modification vers le serveur distant afin de maintenir la cohérence du modèle en fonction des invariants que nous avons définis.

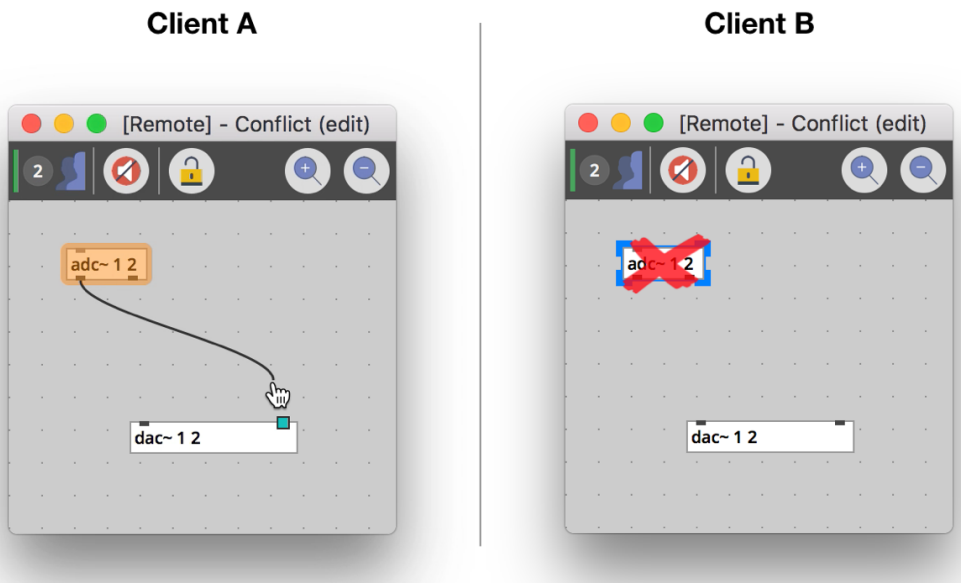

Figure 48 – Capture d'écran du même patch Kiwi chez deux utilisateurs différents connectés au même document et effectuant une modification concurrente sur le patch. À gauche, l'utilisateur A raccorde l'objet *adc~* à l'objet *dac~*. À droite, l'utilisateur supprime l'objet *adc~*.

Si cette logique de validation pourrait être suffisante dans le cadre d'une application mono-utilisateur, nous allons voir à travers un exemple précis qu'il en est en fait tout autrement dans le cadre d'une application multi-utilisateur permettant d'éditer un même document à plusieurs de manière synchrone, et qu'une seconde validation, du côté du modèle présent sur serveur cette fois-ci, est nécessaire pour assurer la cohérence du document. Prenons pour cela le cas concret illustré par la [Figure 48]. Dans cet exemple, un premier utilisateur A, relie la première

sortie de l'objet *adc~* à la deuxième entrée de l'objet *dac~*. Au même moment[144], un autre utilisateur, B, supprime l'objet *adc~* du patch. Au moment où les deux utilisateurs réalisent leur action localement, les deux transactions sont valides, de leur point de vue. Du point de vue de l'utilisateur A, le lien peut tout à fait être créé dans la mesure où les deux objets auquel il est relié résident encore au sein du document. De la même manière, l'utilisateur B peut tout à fait supprimer simplement l'objet puisqu'aucun lien n'y est encore attaché. Les deux transactions sont donc appliquées localement sur chacun des documents puis poussées sur le réseau en vue d'être synchronisées avec le document référent sur le serveur et avec les autres répliques locales chez les différents clients connectés. Ces deux transactions mènent en revanche, du point de vue du document référent situé sur le serveur central à un état incohérent de son modèle de données. En effet, si ces deux transactions étaient appliquées telles quelles, on obtiendrait un document dans lequel un lien serait orphelin, c'est-à-dire qui ne serait rattaché qu'à un seul objet du patch, ce qui a été déclaré sémantiquement non-valide dans le cadre de notre modèle de données. Par quels moyens le serveur de document géré par *flip* permet-il donc de maintenir la cohérence du modèle en cas de conflits de ce type et de faire converger à nouveau les répliques locales vers un état cohérent du document ? *Flip* offre pour cela un mécanisme de validation qui permet d'annuler toute transaction qui génèrerait une incohérence au sein du modèle. Pour continuer avec le même cas d'usage illustré au-dessus, si la transaction de l'utilisateur A arrive en premier au serveur, c'est elle qui sera appliquée, et le lien sera alors effectivement créé. La transaction de l'utilisateur B (suppression de l'objet *adc~*) arrive ensuite au serveur puis un code de validation du modèle est exécuté. Ce code de validation est similaire à un code d'observation. Il consiste à itérer sur chacun des membres du modèle de données afin

[144] Par même moment nous entendons le laps de temps, généralement très court, qui existe entre la synchronisation de toutes les répliques locales. Différentes transactions peuvent néanmoins arriver de manière simultanées au serveur.

de vérifier que tous les invariants sont bien respectés. S'ils le sont, la modification du document est prise en compte et transmise à toutes les répliques locales pour qu'elles puissent se synchroniser avec le document référent. Si au contraire un ou plusieurs invariants du modèle que nous avons définis ne sont pas respectés (ici le fait qu'elle génère un lien relié à un objet supprimé du patch), cette transaction produit un code d'erreur interne sur le serveur qui provoque son annulation. La transaction invalide transmise par un client au serveur est alors rejetée. Le serveur renvoi ensuite cette information de rejet au client concerné qui provoque l'annulation de la transaction qu'il vient de transmettre. Ce dernier retrouve alors un état cohérent du document correspondant à celui du document référent présent sur le serveur (dans ce cas, l'objet *adc~* est à nouveau ajouté au patch dans sa version locale du document).

La bibliothèque *flip* fournit aussi des outils au programmeur qui lui permet de savoir si une transaction a été acceptée ou rejetée par le serveur. Cette information pourrait être utilisée pour notifier l'utilisateur qu'un conflit est survenu à travers l'interface graphique via la mise en place d'une animation. Nous n'avons pas encore exploité cette information au sein de Kiwi, principalement par manque de temps durant le projet, mais ce pourrait être fait dans une prochaine version l'application. En revanche nous verrons dans la prochaine sous-section que des animations ont été ajoutées pour régler d'autres types de conflits d'édition et aider les participants à mieux coordonner leurs actions au sein de l'espace de production.

5.1.6. Mise en place d'animations

Une différence majeure dans la conception d'une application multi-utilisateur, par rapport à une application traditionnelle ne faisant intervenir qu'un seul utilisateur sur un document, est le fait de devoir penser non-seulement à la façon de rendre une action explicite localement, par un retour graphique destiné à l'utilisateur ayant réalisé cette action, mais aussi à la rendre connue et explicite aux autres participants afin qu'ils puissent suivre l'évolution globale du document

et être notifiés des modifications qui y sont opérées. Dans le cas d'une application où l'utilisateur est seul sur un document, ses actions sont de fait connues de lui-même et il n'est donc généralement pas nécessaire que l'interface graphique lui notifie que telle action a été réalisée (mis à part pour indiquer son succès ou son échec quand cela est pertinent). En revanche, lorsqu'un autre utilisateur est présent sur le même document, le retour graphique de l'interface devient pour lui le seul moyen de prendre conscience de l'activité des autres participants. La connaissance des actions réalisées sur un document par d'autres personnes peut être mise en œuvre de différentes manières.

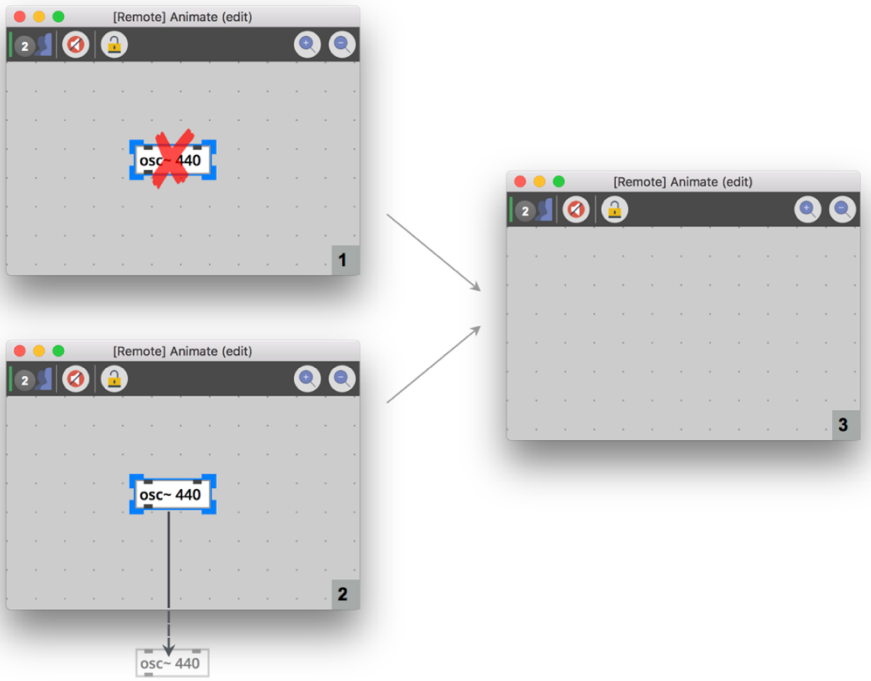

Figure 49 – Capture d'écran montrant deux actions différentes réalisées par un utilisateur sur un même patch menant toutes les deux au même résultat graphique final chez un second utilisateur visualisant ce même document, à droite. En (1) l'objet *osc~* est supprimé du patch, en (2) l'objet est déplacé hors des limites du patch. Dans les deux cas, le patch se retrouve dans l'état final présenté en (3).

Elle peut se faire en temps différé, notamment grâce à la mise en place d'un historique des modifications[145], mais elle doit aussi s'effectuer en temps réel par la mise à jour automatique de l'interface graphique chez tous les participants connectés au document. Dans ce contexte nous nous sommes aperçus par la pratique qu'il pouvait y avoir une confusion dans le type d'action réalisée par un collaborateur du point de vue de l'utilisateur spectateur de la modification [Figure 49]. En effet, prenons par exemple le cas où l'action réalisée est la suppression d'un objet au sein d'un patch par un utilisateur A, l'utilisateur B se retrouvant devant un patch dépourvu de cet objet peut en déduire, à raison, que celui-ci a été supprimé par une autre personne. Mais prenons maintenant le cas où l'utilisateur A déplace simplement l'objet hors de la limite visible du patch. L'utilisateur B, ne voyant plus l'objet au sein du patch, pourrait en déduire de la même façon que l'objet a été supprimé. Comment donc faire en sorte que l'utilisateur B puisse différencier ces deux actions ? Nous aurions pu régler ce problème assez simplement si nous avions choisi de transmettre l'information de déplacement des objets en temps réel aux autres participants, qui auraient alors pu interpréter cette action comme telle, mais cette solution n'a pas été retenue dans la mesure où nous avons jugé qu'elle perturberait plus qu'autre chose l'édition du patch par un excès d'informations inutiles et avons préféré ne transmettre l'information de déplacement ou de redimensionnement des objets aux autres participants uniquement lorsque l'action est terminée. La solution retenue, qui nous a été suggérée par l'équipe *OhmForce* qui l'avait mise en place dans le *OhmStudio*, a été celle d'animer différemment les objets en fonction du type d'action réalisée. Lorsqu'un objet est supprimé au sein du patch, une animation réduisant l'opacité de l'objet est déclenchée pour signaler sa disparition chez les autres collaborateurs. De la même façon, lorsqu'un objet est déplacé, celui-ci n'apparaît pas directement à sa nouvelle position chez les autres participants

[145] Cette fonctionnalité, demandant un temps de développement relativement important, n'est pour l'instant pas encore disponible au sein de la version actuelle de Kiwi.

mais la rejoint progressivement en glissant vers son nouvel emplacement. Ces deux actions sont alors plus facilement différenciables du point de vue de l'utilisateur spectateur de la modification au sein du patch. Ce type de solution graphique entre aussi dans l'espace de coordination dans le sens où elle offre une manière de coordonner les actions des différents participants en leur permettant de prendre plus facilement en considération les actions effectuées par le groupe au sein de l'espace commun de production.

Nous avons abordé dans cette section différents aspects liés à la coordination de l'espace de production relatifs à l'espace de travail partagé du patch. Un autre enjeu relatif à l'espace de coordination se situe au niveau de la gestion de l'accès à ces différents espaces de travail. La section suivante est dédiée plus spécifiquement à ces enjeux.

5.2. Gestion des sessions

La notion de sessions dans un logiciel multi-utilisateur est fondamentale pour coordonner les interrelations entre les personnes et l'accès aux différents espaces de travail. Elles peuvent être vues comme des espaces, dans lesquels les personnes utilisant le logiciel peuvent se retrouver pour travailler ensemble. Nous en avons décrit les principaux enjeux dans la première partie de ce mémoire[146]. Techniquement, une session correspond aux liens qui existent entre un document de référence placé sur un serveur et une réplique locale de ce même document chez plusieurs applications clientes. Du point de vue de l'utilisateur une session correspond simplement à un patch distant auquel il est connecté en temps réel. L'utilisateur peut très bien être seul à travailler dessus, ou bien à plusieurs et ainsi collaborer en éditant et en utilisant le même patch. D'autre part, ces sessions sont aussi persistantes dans le sens où elles sont liées à

[146] Voir section 3.4 (p. 92).

un document physique sur le serveur qui peut être sauvegardé puis rechargé, les rendant donc accessibles par les utilisateurs dans le temps et ouvrant ainsi la voie à une collaboration pouvant aussi s'effectuer de manière asynchrone à travers le logiciel. Dans certains logiciels multi-utilisateurs, les utilisateurs ne peuvent être connectés qu'à une seule session à la fois, c'est par exemple le cas du logiciel *OhmStudio* dans sa version actuelle. Dans les logiciels de patching audio, il est au contraire commun d'ouvrir plusieurs documents à la fois. Kiwi permet donc aux utilisateurs de se connecter à plusieurs sessions indépendantes au sein du logiciel pour ainsi travailler sur différents patchs à la fois avec les mêmes personnes ou des personnes différentes. L'enjeu de la gestion de sessions au sein d'une application multi-utilisateur, tel que nous l'avons formalisé, est aussi lié au contrôle d'accès des documents, c'est-à-dire de déterminer qui a accès à quels documents au sein de l'application. En ce sens elle nécessite donc de pouvoir identifier les personnes au sein de l'application.

La manière de gérer les sessions au sein de l'application Kiwi, l'authentification des utilisateurs ou encore l'accès aux documents distants, autant au niveau de l'interface graphique du logiciel qu'au niveau de l'architecture réseau mise en œuvre, a évolué à plusieurs reprises au cours du développement. Dans les toutes premières versions de Kiwi, il n'était possible de travailler qu'en local sur un patch, il n'y avait donc pour ainsi dire aucune gestion de sessions[147]. La première solution que nous avons mise en place fonctionnait uniquement sur un réseau local et de manière distribuée sur plusieurs machines [Paris & al., 2017]. Elle a été la première à supporter une connexion à des patchs distants au sein de l'application et nous a alors permis d'effectuer les premiers tests de collaboration. Nous revenons plus en détail sur cette solution dans une première partie. Néanmoins, comme nous le verrons, celle-ci n'était pas concluante

[147] Jusqu'à la version *v0.0.3*, sortie en novembre 2016.

sur plusieurs aspects, notamment car elle ne répondait pas aux enjeux de persistance des données, d'accessibilité des documents, d'authentification et de collaboration géographiquement distribuée. Aussi avons-nous dû développer une seconde solution plus robuste, centralisée cette fois-ci, qui nous a alors permis d'offrir plus de flexibilité quant à la manière de gérer les documents distants depuis l'interface graphique de l'application cliente mais aussi l'authentification et le profil des utilisateurs ou encore le stockage des métadonnées liées aux documents. Cette seconde solution, qui est actuellement celle que nous utilisons sera détaillée dans un deuxième temps.

5.2.1. Une première solution décentralisée

La première version de Kiwi permettant de travailler à plusieurs sur un patch fut la *v0.0.3*. Son fonctionnement a en partie été exposé dans [Paris & al., 2017]. L'architecture réseau mise en place s'apparentait alors à une architecture en pair-à-pair. Elle permettait d'éditer un patch à plusieurs de manière collaborative et en temps réel, mais dans la limite d'un réseau local. L'approche consistait à embarquer un serveur de document – fondé sur le modèle de données écrit avec la bibliothèque *flip*[148] – au sein de chaque application cliente de Kiwi. Ce serveur, dont le fonctionnement diffère de celui du serveur actuel, permettait alors de créer des documents Kiwi sur sa propre machine, de les lister et de s'y connecter pour les éditer et les utiliser de manière collaborative. Mais sa particularité était surtout qu'il intégrait un composant logiciel permettant d'exposer les documents locaux dont il avait la gestion aux autres personnes présentes sur le réseau par le biais d'une technologie de *multicast*[149] permettant la découverte dynamique de services (possibilité de rejoindre un document sur machine distante). De la même

[148] Voir sous-section 4.2.1 (p. 127) de ce document.

[149] Le multicast est une topologie de réseau dans laquelle un émetteur peut diffuser une information vers un groupe de récepteurs. Cette solution était soutenue par l'API de *flip*.

façon, ce serveur pouvait aussi *écouter* les services proposés par les autres serveurs actifs sur le réseau pour *découvrir* et offrir à l'utilisateur des documents distants en édition, situés physiquement sur la machine d'un autre collaborateur.

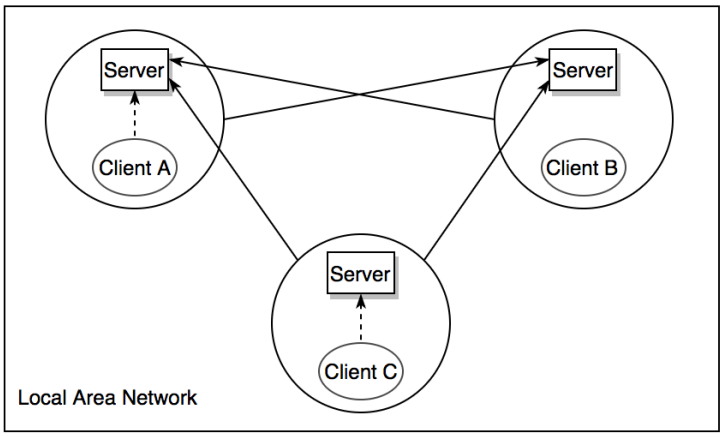

Figure 50 – Schéma présentant l'architecture réseau en place à la version *v0.0.3* du logiciel Kiwi et décrite dans [Paris & al., 2017]. Chaque client dispose d'un serveur embarqué au sein de l'application à partir duquel il peut se connecter à des documents sur sa propre machine. Les utilisateurs peuvent aussi se connecter à des documents exposés par le serveur embarqué sur d'autres applications clientes présentes sur le réseau local (*Local Area Network*). Les flèches en pointillés symbolisent des connexions qui s'effectuent au sein d'une même machine locale, les autres représentent des connexions à distance à des documents sur le réseau local.

Dans le schéma ci-dessus, illustrant cette architecture, trois utilisateurs sont connectés à un même réseau local [Figure 50]. Ces trois utilisateurs exécutent chacun une application cliente embarquant un serveur permettant de se connecter à des documents. L'utilisateur de l'application cliente A est connecté à un document sur son propre serveur et à un document géré par l'application cliente B, le client B se connecte à un document chez le client A, il n'est connecté à aucun document sur son propre serveur mais gère un document auquel accèdent les clients A et C. Le client C est quant à lui connecté à un document sur son propre serveur et à un document situé chez les clients A et B.

Pour gérer l'interface de connexion aux documents au sein de l'application cliente un composant graphique spécifique nommé *Document Browser* a été créé.

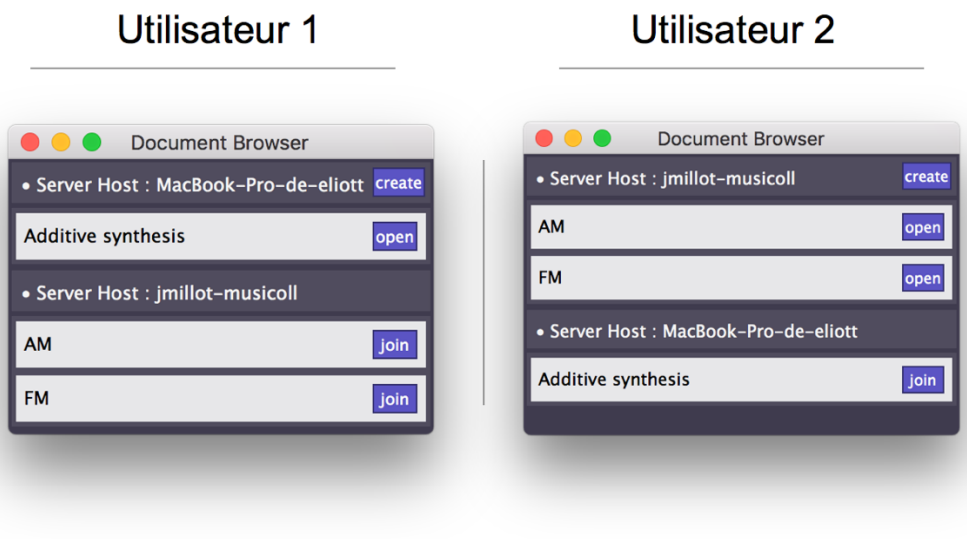

Figure 51 – Capture d'écran des fenêtres nommées *Document Browser* telles qu'elles pouvaient se présenter, à la version *v0.0.3* de l'application Kiwi, chez deux utilisateurs différents reliés par l'application à travers le même réseau local. À gauche on peut voir qu'un premier utilisateur dispose d'un document sur sa machine, nommé *Additive synthesis* ; à droite un second utilisateur dispose de deux documents (nommés *AM* et *FM*).

Dans cette interface plutôt sommaire [Figure 51], les utilisateurs pouvaient visualiser et ouvrir les documents situés sur leur machine (grâce au bouton *open*), mais aussi les documents joignables sur les autres machines connectées au même réseau local qu'eux (grâce au bouton *join*). Le bouton *create* permettait quant à lui de créer des documents, sur la machine locale, que les autres personnes pouvaient ensuite rejoindre. Elle permettait aussi d'offrir en ce sens une première visualisation des utilisateurs et une conscience de groupe au sein de l'application en affichant sous forme de liste le nom de chaque ordinateur présent sur le réseau. L'avantage de cette première architecture est qu'elle permettait une facilité d'accès aux documents et nous a permis de tester rapidement les aspects collaboratifs de l'application déjà mis en place. Elle comportait néanmoins un certain nombre de limites que nous exposons ici.

Le premier inconvénient était que les documents étaient stockés localement sur la machine de chaque utilisateur. De ce fait, lorsqu'un utilisateur se déconnectait du réseau local, ou était simplement absent, il rendait indisponibles aux autres utilisateurs les documents dont il avait la gestion. Or, parmi les spécifications initiales données à l'application Kiwi, nous voulions que celle-ci puisse offrir une accessibilité constante des documents aux utilisateurs. Une autre limite ce système était qu'il ne permettait pas de disposer d'une autorité qui fixe les identités des utilisateurs présents sur le réseau et par extension sur les différents documents. La connexion d'un utilisateur à un document *flip* se fait entre autres en fournissant un identifiant unique qui permet de le distinguer des autres utilisateurs. Cet impératif permet au système *flip* de différencier les modifications apportées par chaque utilisateur au sein du document et de contrôler ainsi son édition concurrente. Il est nécessaire aussi à la mise en place du système de sélection multi-utilisateur dans la mesure où ces identifiants sont utilisés pour distinguer la sélection locale d'un utilisateur de celle des autres au sein du document. La solution alors mise en place était d'attribuer des identifiants à chaque utilisateur de manière aléatoire. Grâce à ce système, chaque utilisateur disposait d'un identifiant propre sur le réseau, néanmoins les risques de collision – le fait que plusieurs utilisateurs sur le réseau disposent du même identifiant – étaient possibles dans la mesure où aucune instance ne garantissait l'unicité de ces identifiants et des incohérences pouvaient alors survenir. D'autre part, cette solution ne convenait pas au stockage des métadonnées liées aux documents (*e.g.* nom du document, date de création), et ne permettait pas non-plus de contrôler un accès authentifié aux documents – c'est-à-dire d'être sûr que c'est bien un utilisateur précisément qui accède au document et pas un autre. La dernière limite liée à cette architecture que nous exposerons dans ce cadre, qui est sans doute aussi la plus importante, était qu'elle ne résolvait pas la problématique de l'édition distribuée géographiquement dans la mesure où l'accès au document n'était permis qu'au sein d'un réseau local.

Ce système a donc été revu par la suite et une seconde architecture, cette fois-ci centralisée, a été créée.

5.2.2. Passage à une gestion de sessions centralisée

Une centralisation est nécessaire si l'on veut garantir l'unicité des informations et coordonner l'accès à des ressources communes sur plusieurs sites distribués géographiquement. Pour cela, nous avons donc dû abandonner la solution technique mise en place initialement. Depuis la version *v0.1* de Kiwi, qui a été présentée pour la première fois à l'occasion des Journées d'Informatique Musicale en Mai 2017, l'application cliente de Kiwi n'embarque plus de serveur, et la technologie de *multicast* a été retirée au profit d'une architecture réseau plus standard, permettant aux utilisateurs de se connecter à un serveur centralisé. Cette dernière architecture [150], que nous maintenons encore actuellement dans Kiwi, nous permet de sauvegarder l'ensemble des documents créés par les utilisateurs de l'application au même endroit et de maintenir un accès simplifié à ces documents distants depuis des sites géographiquement distribués, à travers le réseau internet.

Ce n'était néanmoins pas le seul enjeu du basculement vers une architecture serveur centralisée. Pour permettre aux utilisateurs de retrouver leurs documents en ligne facilement, il était aussi nécessaire d'organiser le stockage de métadonnées relatives aux documents tels qu'un nom et un identifiant unique par document. Nous souhaitions aussi que les utilisateurs puissent disposer d'informations supplémentaires au sein de l'interface pour chaque document distant, qui leur permettent de connaître sa date de création, savoir par exemple qui l'a créé ou encore qui l'a ouvert dernièrement. Pour cela il nous fallait aussi pouvoir stocker des informations

[150] Que nous avons décrite partiellement à la partie 4.2.3 (p. 135) de ce document.

relatives aux utilisateurs de l'application. D'autre part, nous voulions que les utilisateurs de Kiwi puissent être identifiés de manière unique par le système, qu'ils puissent donc créer un compte depuis l'application en renseignant des informations sur leur profil (par exemple leur nom d'utilisateur et leur email), afin que le système puisse leur attribuer un identifiant garanti comme unique, et qu'ils puissent être authentifiés comme tel auprès du serveur et au sein des documents auxquels ils se connecteraient.

Si la bibliothèque *flip* nous permet de gérer la connexion aux documents et leur sauvegarde, elle ne dispose pas d'outils logiciels permettant la gestion d'informations plus générales relatives aux documents ou encore à la gestion des utilisateurs au sein d'une application. Pour répondre à ces différents enjeux, il nous a donc fallu développer une solution parallèle au serveur de document *flip* qui nous permette d'organiser le stockage de ces informations et permettre de les requêter depuis l'application cliente afin de les utiliser notamment en les affichant à l'utilisateur.

La réponse technique à ces enjeux a été de développer une API Web spécifique à Kiwi qui permet notamment la gestion des comptes utilisateurs, l'authentification des utilisateurs ou encore l'organisation et la gestion des documents. Pour cela nous avons fait le choix de recourir à la technologie *Node.js*[151], fondée sur le langage *JavaScript*, couplée avec une base de données *MongoDB*[152]. Le choix de la technologie *Node.js* a été fait notamment car elle nous paraissait plus accessible du fait que nous possédions déjà une petite expérience dans ce langage et le

[151] https://nodejs.org/fr/ [consulté le 12/09/18].

[152] *MongoDB* est une base de données de type *NoSQL* permettant la gestion d'objets structurés au format BSON (Binary JavaScript Object Notation) : https://www.mongodb.com/ [consulté le 12/09/18].

développement Web, mais aussi car elle vient avec l'écosystème *NPM*[153], qui est constitué d'énormément de modules préconçus pour ce genre d'usage, ce qui a permis d'accéléré le temps de développement[154].

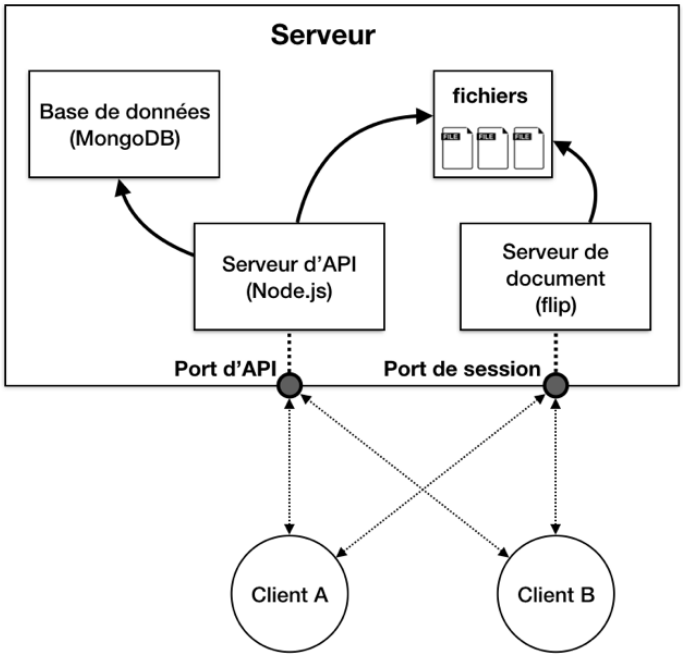

Figure 52 – Schéma représentant l'architecture centralisée mise en place à partir de la version *v0.1* de l'application Kiwi. Le serveur de Kiwi est composé de trois processus différents : la base de données (technologie *MongoDB*), le serveur Web exposant l'API (technologie *Node.js*) et le serveur de document (technologie *flip*). Le server d'API est connecté à la base de données et, comme le serveur de document, a accès aux fichiers de la machine. Ils exposent tous deux un port différent à l'extérieur, auquel se connectent les applications clientes de Kiwi. Ici, deux clients sont connectés au serveur (sur les deux ports).

Le schéma présenté ci-dessus expose l'architecture du serveur de Kiwi telle qu'elle est actuellement en place. Elle se compose de trois processus différents tournant en parallèle [Figure 52]. La base de données sert au stockage des informations (relatives aux documents et

[153] *Node Package Manager* est le plus gros gestionnaire de dépôt de paquets, tous langages confondus : https://www.npmjs.com/ [consulté le 06/09/18].

[154] Ces modules nous ont permis de développer un serveur, de gérer l'interface avec la base de données ou encore de gérer l'authentification des utilisateurs, plus facilement que si nous avions utilisé le langage C++ où tout aurait dû être recréé.

comptes utilisateurs). Le serveur de document permet la connexion des utilisateurs aux sessions, c'est-à-dire aux documents disponibles sur la machine distante, et gère aussi l'édition concurrente des documents par les utilisateurs, c'est celle que nous avons décrite à la sous-section 4.2.3 (p. 135) de ce document. Le troisième processus est un serveur exposant une API Web permettant d'effectuer des opérations en ligne en transmettant des requêtes au format HTTP[155]. L'API de Kiwi[156] a été développée suivant les principes REST[157]. Ces principes sont devenus très populaires dans l'élaboration de services Web. Ils permettent d'effectuer plusieurs opérations à partir d'une même URL (ou *endpoint* dans le jargon REST), grâce à différentes méthodes HTTP matérialisées par des verbes spécifiques. On utilise ainsi une requête *GET* pour lire ou acquérir une ressource (*e.g.* demander la liste des documents ou télécharger un document précis), la méthode *POST* pour la création d'une nouvelle ressource (*e.g.* un nouvel utilisateur ou un nouveau document), la méthode *PUT* pour effectuer une mise à jour d'une ressource (*e.g.* le changement de nom d'un document) et enfin la méthode *DELETE* pour tout ce qui relève de la suppression de ressource (*e.g.* supprimer un document ou un utilisateur)[158].

Au sein de l'API Web de Kiwi, certaines requêtes nécessitent d'être authentifiées pour renvoyer une réponse valide. Cette authentification se fait par l'intermédiaire d'un jeton de sécurité au format *JWT*[159] stocké localement par chaque application cliente. À chaque fois que l'utilisateur

---

[155] L'*Hypertext Transfer Protocol* (protocole de transfert hypertexte) est un protocole de communication client-serveur développé pour le Web.

[156] Le code du serveur *Node.js* que nous avons développé en *open-source* est disponible dans un répertoire séparé https://github.com/Musicoll/kiwi-node-server [consulté le 12/09/18].

[157] *REpresentational State Transfer* (Transfert d'état représentatif). REST est un ensemble de principes architecturaux ou de lignes directrices permettant de construire des systèmes distribués [Fielding, 2000].

[158] La liste de l'ensemble des opérations réalisables en ligne à travers l'API de Kiwi est documentée à l'adresse suivante : https://musicoll.github.io/kiwi-node-server/ [consulté le 12/09/18].

[159] Le *JWT* (*JSON Web Token*) est un standard JSON (JavaScript Object Notation) ouvert, défini par la RFC-7519, qui permet la création et l'échange de jetons (*tokens*) d'accès sécurisés entre plusieurs parties. La sécurité de l'échange se fait par la vérification de l'intégrité des données que contient le jeton à l'aide d'une signature numérique qui s'effectue par l'algorithme *RSA* ou *HMAC* puis par une validation de son contenu.

veut accéder à une ressource protégée, il doit fournir avec chaque requête le jeton d'accès qui lui a été donné préalablement lors de son authentification auprès du serveur. Dans le cadre de Kiwi nous utilisons pour l'instant trois jetons différents. Le premier est généré quand l'utilisateur s'enregistre avec son nom d'utilisateur et son mot de passe auprès de l'API Web. Le jeton d'accès qui lui est alors renvoyé est stocké localement sur la machine puis transmis ensuite avec chaque requête nécessitant une authentification auprès du serveur. Un second type de jeton a été mis en place. Pour ouvrir un document sur le serveur distant la procédure est la suivante : en cliquant sur un des documents dans la liste présentée par le *Document Browser*[160], l'utilisateur envoie une première requête à l'API en demandant d'accéder à une ressource spécifique. Si le jeton passé en en-tête est valide et que l'utilisateur dispose des droit suffisants pour accéder à cette ressource, il génère un jeton d'accès sécurisé comportant des informations telles que la version de l'application et l'identifiant de l'utilisateur en demandant l'accès et le renvoie comme réponse. L'application cliente, disposant désormais du jeton d'ouverture pour le document en question peut alors effectuer une nouvelle requête, cette fois-ci au serveur de document, en transmettant le jeton qu'il vient de recevoir de la part de l'API dans les métadonnées de connexion. Le serveur de document vérifie alors ce jeton d'accès et entame l'ouverture d'une session pour ce document si tous les paramètres sont bons. Un troisième type de jeton est employé lorsque l'utilisateur a perdu son mot de passe. Dans ce cas l'utilisateur peut demander de le réinitialiser à partir de l'interface graphique du logiciel en saisissant son adresse électronique. Celui-ci reçoit ensuite par courrier électronique un jeton de sécurité qu'il devra alors fournir avec son nouveau mot de passe lors de sa réinitialisation pour pouvoir s'enregistrer à nouveau auprès du serveur qui lui fournira alors un nouveau jeton d'authentification.

[160] Cette interface sera décrite plus en détails par la suite (5.2.2.3, p. 206).

La topologie du réseau ayant changé, mais aussi la manière de se connecter aux documents distants, nous avons donc dû adapter aussi l'interface graphique de l'application cliente de l'application pour permettre aux utilisateurs de configurer les options liées au réseau, de s'enregistrer et de s'authentifier auprès du serveur central, mais aussi d'afficher et gérer les documents distants afin de les rejoindre. Nous consacrons les trois prochains points à ces différents composants afin d'exposer les différentes opérations permises à l'utilisateur au sein de l'application.

5.2.2.1. Configuration des options liées au réseau

La fenêtre de préférences générales de Kiwi permet d'accéder aux différents réglages liés aux paramètres de connexion au réseau.

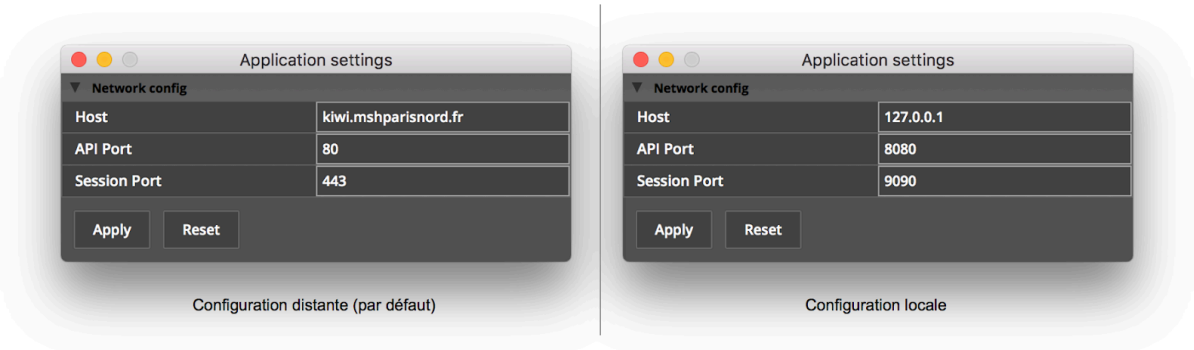

Figure 53 - Capture d'écran de la fenêtre de réglages des préférences de l'application Kiwi exposant deux configurations réseau différentes : distante (à gauche) ou locale (à droite).

Il est possible de spécifier à partir de celle-ci :

- L'hôte auquel doit se connecter l'application sous la forme d'une adresse IP[161] (sur le réseau local ou global) ou d'un nom de domaine - une résolution DNS[162] sera alors effectuée dans ce cas par l'application.
- Le port de connexion à l'API de Kiwi.
- Le port de connexion au serveur de document Kiwi.

Dans le cadre du projet MUSICOLL, la société partenaire *OhmForce* a mis à notre disposition un serveur que nous utilisons à la fois comme serveur de développement et comme serveur de production pour l'application Kiwi[163]. L'application est configurée par défaut pour s'y connecter automatiquement au premier lancement de l'application. La plupart des utilisateurs n'ont donc pas besoin de modifier ces réglages. Cependant, nous avons tenu à maintenir un accès direct à ces réglages pour offrir plus de flexibilité à de futurs contributeurs externes qui rejoindraient le projet ou des utilisateurs avancés, mais surtout pour nous faciliter la tâche durant le processus de développement. Cette flexibilité permet notamment :

- De se connecter au serveur de production disponible dans le cadre du projet MUSICOLL – dans la configuration proposée par défaut exposée à gauche de la [Figure 53].
- Se connecter à un autre hôte distant ou sur un autre port pour tester de nouvelles fonctionnalités avec un serveur différent sans impacter le travail des utilisateurs de Kiwi sur le serveur de production.

---

[161] Internet Protocol

[162] Le *Domain Name System*, ou système de noms de domaine en français, est le service informatique distribué utilisé notamment pour traduire les noms de domaine Internet en adresse IP.

[163] Ce serveur est hébergé sur la plateforme Amazon sous la forme d'une instance *EC2* et rendu disponible à l'adresse http://kiwi.mshparisnord.fr/ [consulté le 20/09/18] par une redirection DNS effectuée par la MSH Paris-nord qui facilite ainsi son accès.

- D’héberger son propre serveur sur sa machine et s’y connecter depuis Kiwi – comme dans la configuration exposée à droite de la [Figure 53].
- De se connecter à un serveur lancé sur une machine présente sur un réseau local, dans le cas où nous n’aurions pas accès à internet.

5.2.2.2. Enregistrement et authentification

Si l’utilisateur n’a pas modifié les réglages réseaux d’origine ou s’ils sont configurés correctement, il peut alors se rendre dans le menu général *Account* de Kiwi pour se créer un compte puis s’authentifier auprès du serveur central.

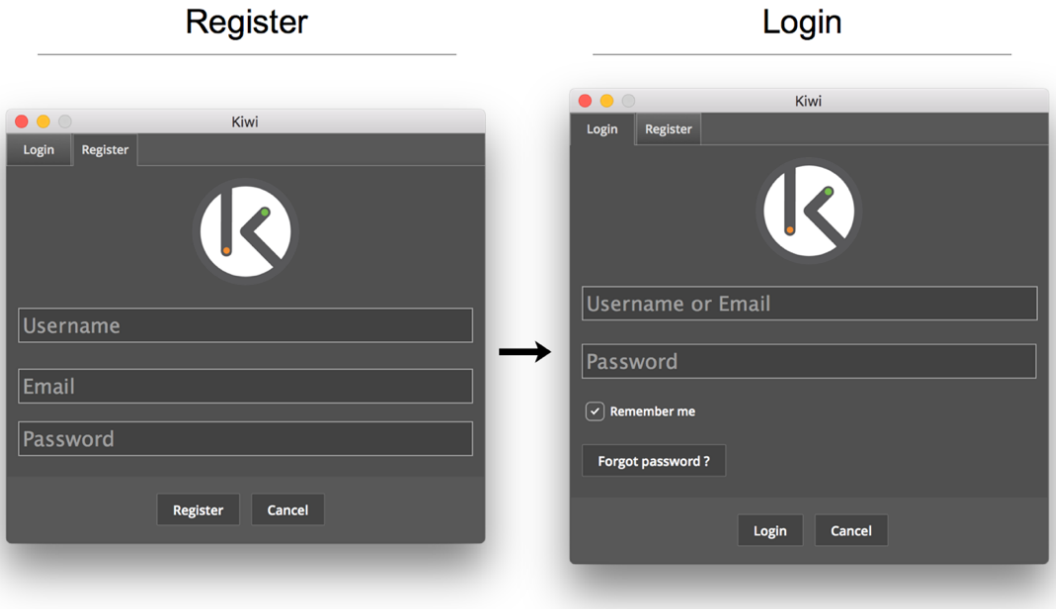

Figure 54 – Capture d’écran exposant deux fenêtres de l’application Kiwi correspondant aux deux étapes nécessaires à l’authentification d’un nouvel utilisateur au sein de Kiwi. À gauche la fenêtre *Register* offre un formulaire permettant à l’utilisateur de créer son compte. À droite la fenêtre de *Login* affiche un formulaire d’authentification auprès du serveur.

L’étape de création du compte se fait par l’intermédiaire de l’onglet *Register* de la fenêtre d’authentification de Kiwi exposée à gauche de la [Figure 54], qui propose un formulaire avec trois champs requis :

- Le nom de l'utilisateur, qui peut être son prénom ou un surnom, c'est le nom que verront s'afficher les autres personnes connectées à l'application pour se référer à lui.
- L'adresse électronique. Celle-ci doit être valide pour que le serveur puisse communiquer avec l'utilisateur.
- Le mot de passe (plus de 8 caractères).

Si une erreur survient – par exemple si le serveur ne répond pas, le nom d'utilisateur existe déjà, l'adresse est incorrecte ou le mot de passe ne convient pas – une notification est affichée à l'utilisateur sous la forme d'une bannière rouge en haut de la fenêtre et/ou dans la console de l'application. Si tous les champs sont correctement remplis, l'utilisateur est notifié que tout s'est bien passé et reçoit alors un message électronique comportant une clef d'activation de son compte sous la forme d'un lien sur lequel il doit cliquer pour confirmer son inscription. Une fois cette étape réalisée, l'utilisateur peut alors s'authentifier auprès du serveur à partir de l'onglet *login* de la fenêtre d'authentification [Figure 54]. Il doit alors fournir, au choix son adresse électronique ou le nom d'utilisateur qu'il a choisi lors de sa création de compte, ainsi que son mot de passe. Si tout s'est bien passé, l'utilisateur est notifié par un message de succès qui s'affiche dans la même fenêtre, lui indiquant qu'il est désormais authentifié auprès du serveur. La fenêtre *Document Browser*, que nous aurons l'occasion de détailler plus après, est alors rafraîchie et affiche la liste des documents en ligne qu'il peut désormais rejoindre. Le formulaire de *login* possède aussi un champ « *remember me »* de type interrupteur, qui a pour effet, s'il est activé, de sauvegarder les réglages relatifs aux informations d'authentification de l'utilisateur afin que celui-ci n'ait pas à les ressaisir à chaque nouveau lancement de l'application et puisse maintenir sa connexion avec le serveur.

Dans le cas où l'utilisateur aurait perdu son mot de passe, le formulaire de *login* intègre une option accessible en cliquant sur le bouton « *Forgot password ?* » [Figure 54], qui lui permet de le changer.

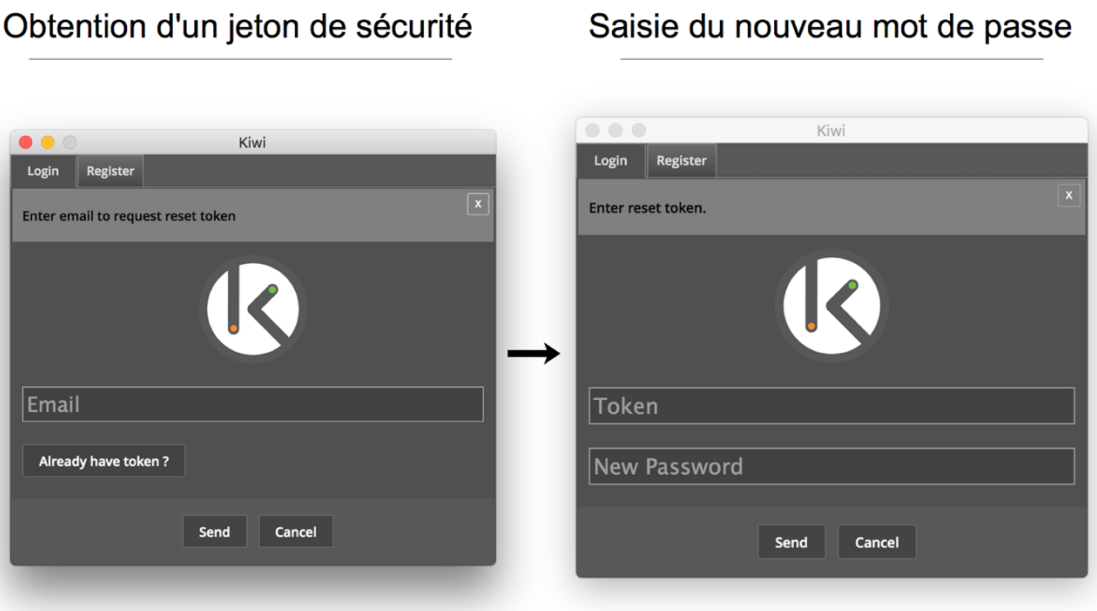

Figure 55 – Capture d'écran exposant la même fenêtres de l'application Kiwi, correspondant aux deux étapes nécessaires à la reconfiguration d'un nouveau mot de passe pour un utilisateur. À gauche un formulaire permet à l'utilisateur de saisir son adresse électronique. La fenêtre de droite invite l'utilisateur à saisir son nouveau mot de passe accompagné d'un jeton de sécurité fourni à l'étape précédente.

La procédure de modification est la suivante. L'utilisateur doit tout d'abord fournir son adresse électronique dans une première fenêtre [Figure 55, à gauche]. Si elle est valide, le serveur lui envoie par courrier électronique un jeton de sécurité qu'il doit alors copier dans la fenêtre de droite au moment de saisir le nouveau mot de passe de son choix [Figure 55, à droite]. Si la reconfiguration du mot de passe est validée par le serveur, l'application renvoie l'utilisateur vers l'onglet *login* afin qu'il puisse de nouveau s'authentifier auprès du serveur.

5.2.2.3. Gestion des documents distants

Contrairement à la première version de l'interface, exposée à la [Figure 51, p. 194], la fenêtre nommée *Document Browser* n'affiche désormais plus le nom de l'ordinateur des participants. Quand un utilisateur démarre l'application pour la première fois, qu'il n'est pas connecté à internet ou qu'il n'est pas authentifié auprès du serveur, la fenêtre du *Document Browser* est vide et le texte *Offline* est affiché à côté de l'icône de dossier.

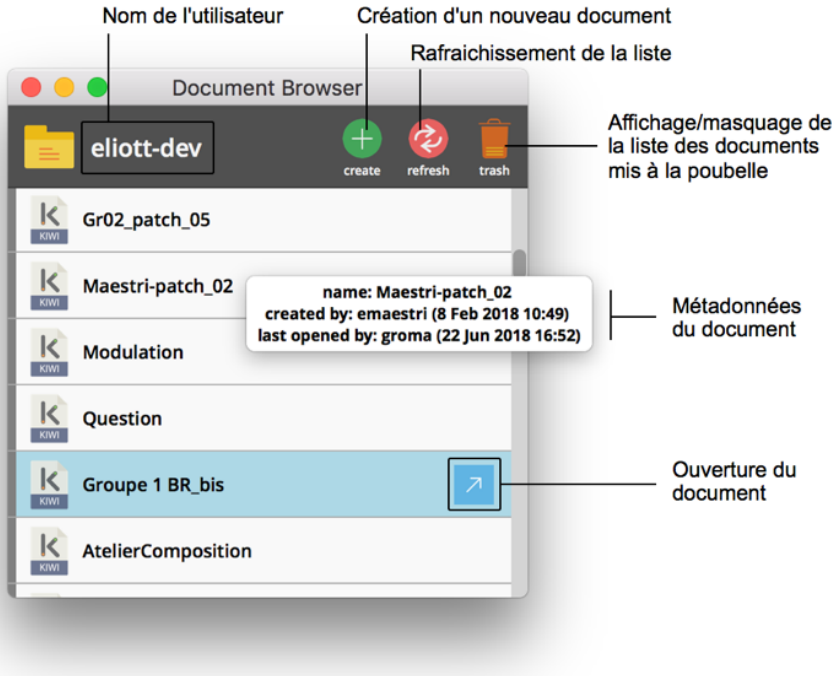

Figure 56 – Capture d'écran légendée de l'interface présentée par la fenêtre *Document Browser* de l'application Kiwi à la version *v1.0.2*, lorsque l'utilisateur est authentifié auprès du serveur.

Lorsque l'utilisateur est connecté à l'application et qu'il est correctement authentifié, son nom, tel qu'il est connu du serveur, s'affiche en haut à gauche [Figure 56]. Cette fenêtre intègre alors un composant graphique présentant la liste des documents présents sur le serveur central qui est mise à jour automatiquement et à intervalle régulier par l'application, mais qui peut aussi être rafraîchie au besoin manuellement par un bouton. L'utilisateur peut créer un nouveau

document sur le serveur en pressant le bouton *create* qui ouvre alors une fenêtre temporaire lui permettant de saisir le nom qu'il souhaite donner au nouveau patch. Ce document apparaît alors avec les autres dans la liste chez tous les utilisateurs connectés au serveur qui peuvent alors l'ouvrir simplement en double-cliquant sur l'élément. Pour garantir que l'accès au document est autorisé et permettre son ouverture par l'utilisateur, l'application cliente doit effectuer une première requête au serveur en demandant une clef d'ouverture correspondant au document en question à l'API de Kiwi[164]. Cette clef est ensuite transmise au serveur de document par l'application qui vérifie sa validité avant de permettre la connexion. Ce système est en fait à la base de l'élaboration d'un système plus complexe qui n'a pour l'instant pas pu être développé dans le cadre du projet. Pour l'instant, tous les documents présents sur le serveur central sont accessibles à l'ensemble des utilisateurs. La clef d'activation n'est déclarée invalide que dans le cas où un utilisateur ne serait pas correctement authentifié auprès du serveur. On peut néanmoins imaginer à terme que ce système puisse gérer des cas plus complexes en attribuant des rôles aux utilisateurs en fonction des différents documents et filtrer leur accès en fonction grâce à ce même procédé.

[164] L'API renvoi une clef valide si l'utilisateur est authentifié auprès du serveur, sinon cela provoque l'envoi d'un code d'erreur qui empêche simplement l'utilisateur d'accéder au document en l'invitant à s'authentifier à nouveau.

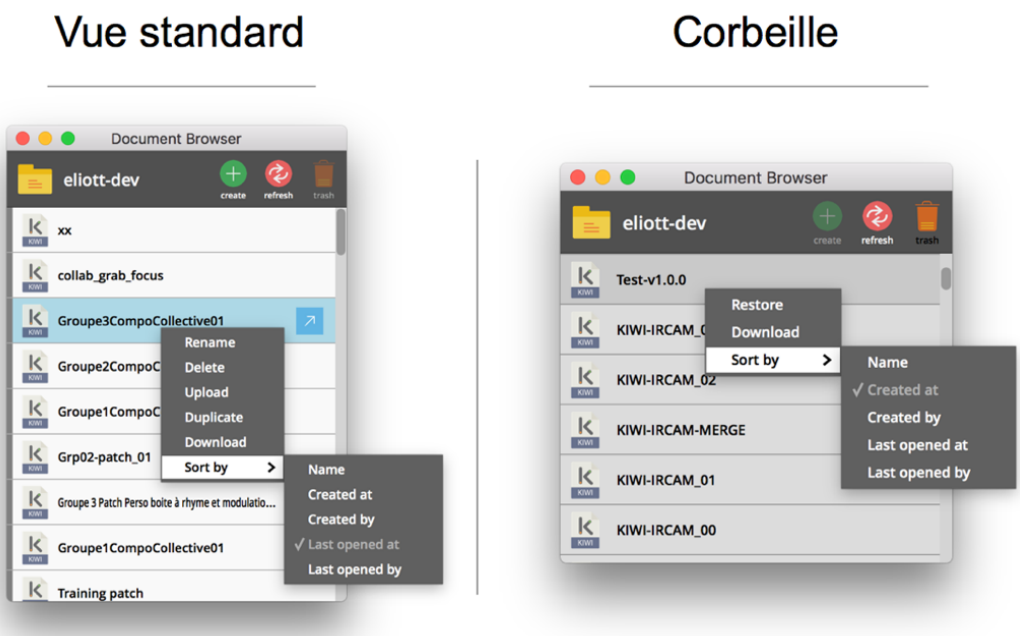

Figure 57 – Capture d'écran de la fenêtre *Document Browser* de Kiwi dans deux états différents : à gauche la vue standard, et à droite la vue de la corbeille. Elle montre aussi le menu contextuel à partir duquel et en fonction de l'état de la fenêtre, il est possible d'effectuer certaines opérations.

L'interface fournit aussi quelques métadonnées sous la forme d'une fenêtre flottante s'affichant au survol d'un élément par la souris. Ces métadonnées permettent notamment de savoir qui a créé le document, qui l'a ouvert pour la dernière fois et à quel moment [Figure 56]. En ce sens, elle offre une première réponse à la problématique liée au fait de fournir une conscience de la visite (*visitor awareness*) au sein d'une interface multi-utilisateur, en permettant aux utilisateurs de savoir *qui* a visité un document en dernier [Gellersen & Schmidt, 2002].

L'interface de la fenêtre *Document Browser* permet aussi d'effectuer certaines opérations sur les documents à partir d'un menu contextuel qui autorise le renommage, la suppression, la duplication, le téléversement, le téléchargement de documents [Figure 57, à gauche] ou encore la restauration après suppression [Figure 57, à droite]. La suppression d'un document permet de le placer dans une corbeille virtuelle commune, il n'est alors pas supprimé effectivement

mais simplement mis de côté. L'interface permet alors de visualiser les documents placés par les utilisateurs à la corbeille et de les restaurer au besoin [Figure 57, à droite].

Le composant graphique exposé au sein de cette fenêtre est donc un élément central et fondateur de l'interface multi-utilisateur de Kiwi, donnant une porte d'accès à la collaboration sur les documents. Il permet d'obtenir des informations et de gérer les différents canaux de communications et de production que représentent les documents joignables sur le serveur central. Néanmoins il souffre encore de nombreuses incomplétudes, surtout dues à un manque de temps durant le projet et à une certaine complexité de mise en œuvre.

Actuellement les seuls fichiers visibles et manipulables au sein de cette fenêtre sont les documents de type patch, mais on pourrait très bien imaginer à terme que l'on puisse visualiser, accéder et manipuler d'autres types de documents ou de ressources en ligne dans la même interface, comme des images ou des fichiers sons, mais un développement préalable doit être prévu au niveau du noyau logiciel pour supporter ce genre de ressources au sein du patch. D'autre part, le navigateur de documents n'intègre pour l'instant qu'un seul niveau hiérarchique, il n'existe pas de concept de dossier ou de sous-dossier. Il peut donc être difficile de s'y retrouver, d'autant plus que les documents se trouvent tous dans un espace qui est commun et accessible à tous les utilisateurs. En attendant de trouver le temps pour réfléchir en profondeur aux spécifications à donner à un espace privé, et de développer ce système nous avons donc choisi d'offrir des petites améliorations pour mieux s'y retrouver au sein de la liste. Le menu contextuel va dans ce sens en proposant de configurer la manière dont sont affichés les documents en permettant la configuration d'une option de tri en fonction du nom, de la date de création ou de dernière ouverture, ou encore de l'auteur qui a créé ou ouvert en dernier les documents. Cette fonctionnalité était primordiale à mettre en place avant de livrer la première

version utilisable de manière publique dans la mesure où cela a permis de retrouver plus facilement ce que l'on cherche au sein de l'interface.

5.2.3. Notifications et gestions des erreurs de connexion

Les erreurs survenant au cours de l'exécution d'un collecticiel peuvent être multiples et de différentes natures, il s'agit alors de faire en sorte que l'utilisateur ait une bonne compréhension de ces erreurs en les lui notifiant correctement, mais aussi de lui proposer des solutions alternatives lorsqu'elles surviennent (*e.g.* passage du document en mode hors-ligne lors d'une déconnexion au serveur par exemple). Ce procédé est souvent désigné par le terme de dégradation gracieuse, par opposition à une dégradation disgracieuse qui réduit la qualité de service sans chercher à minimiser l'impact sur l'utilisateur.

L'interface du logiciel est la seule à pouvoir faire transparaître les erreurs qui peuvent survenir lors de l'exécution du programme. C'est donc son rôle de notifier l'utilisateur et de lui proposer des solutions alternatives pour les gérer. De manière générale, les erreurs sont reportées dans la console de l'application. Ces messages peuvent être des notifications de succès, par exemple : « *connexion établie à l'hôte kiwi.mshparisnord.fr* », des messages d'avertissement comme « *La version de l'application courante n'est pas compatible avec la version installée sur le serveur, veuillez télécharger la dernière version* » ou encore d'erreur lorsque la connexion au serveur ou à un des documents sur le serveur a été perdue, qu'un document ne peut pas être converti, téléchargé ou téléversé. Nous noterons dans ce cadre que que c'est la fenêtre *Document Browser* qui reflète le mieux le statut courant de la connexion de l'application avec le serveur actuellement en affichant, comme nous l'avons vu dans la sous-section précédente, soit le texte « *offline* », soit le nom de l'utilisateur ainsi que la liste des documents joignables, en fonction du statut courant de la connexion.

Le concept de dégradation gracieuse se définit généralement comme la capacité d'un ordinateur, un logiciel ou encore un réseau à maintenir des fonctionnalités limitées même lorsqu'une majeure partie est rendue inopérante. Un cas de dégradation de service qu'il nous a fallu gérer au sein de Kiwi est le cas où l'utilisateur subit une déconnexion depuis le serveur, alors qu'il est en train d'éditer un document distant, c'est-à-dire la déconnexion à une session. Dans les premières versions de l'application aucune notification ne lui était adressée. Cela engendrait alors des problèmes notoires tels qu'une perte du travail réalisé suite à la déconnexion, vu que l'application lui autorisait à continuer d'éditer le document en ne lui indiquant pas qu'il n'était plus synchronisé avec le serveur distant et donc que ses modifications ne seraient pas enregistrées. Cette dégradation de service pouvait alors être qualifiée de disgracieuse dans la mesure où elle ne s'adaptait pas aux erreurs, et pire, qu'elle pouvait être génératrice de problèmes de synchronisation. Des développements ont alors été entrepris pour régler la situation.

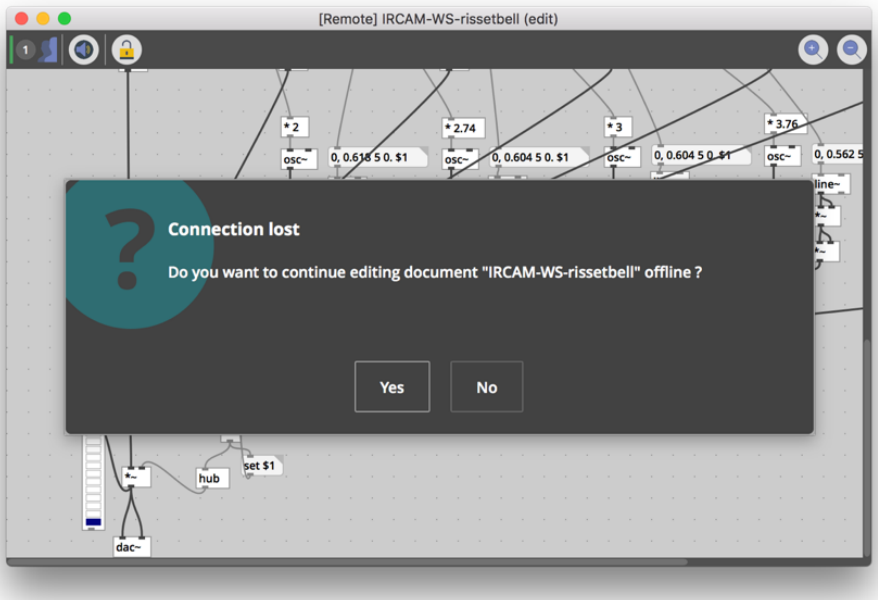

Figure 58 – Capture d'écran de la fenêtre temporaire exposée à un utilisateur de Kiwi suite à une perte de connexion au document distant, lui proposant alors de passer dans un mode d'édition hors-ligne.

Lorsque l'utilisateur subit une déconnexion depuis le serveur alors qu'il est en train d'éditer un document distant, une fenêtre de dialogue s'affiche désormais au-dessus du patch lui notifiant de la situation et lui proposant de continuer à éditer le document en mode hors-ligne [Figure 58]. La dégradation de service est alors considérée comme plus gracieuse dans la mesure où l'utilisateur perd la faculté de travailler en ligne – et donc le fait travailler avec d'autres personnes ou encore de sauvegarder automatiquement ses modifications – mais peut continuer son activité au sein du patch localement. Si l'utilisateur ne souhaite pas conserver son patch, la fenêtre se ferme, sinon celle-ci adapte son apparence pour indiquer que le patch est désormais local (le nombre d'utilisateurs disparaît de la barre d'outils et le texte de la barre de titre n'indique plus le texte « [Remote] »). Comme tout autre document local, celui-ci peut ensuite être sauvegardé sous la forme d'un fichier sur la machine de l'utilisateur.

Cette solution représente une première étape mais n'est bien sûr pas encore optimale dans la mesure où l'application gère actuellement uniquement le cas de la déconnexion mais pas celui de la reconnexion. Kiwi ne fournit en effet encore aucun moyen automatisé à l'utilisateur pour effectuer l'opération inverse qui serait, lors d'une reprise de la connexion, de synchroniser à nouveau le patch local avec le patch distant. Les modifications apportées localement et de manière hors-ligne au document distant devraient alors être fusionnées avec la version référente du document en ligne afin de proposer de reprendre une édition en temps réel synchronisée avec celle-ci.

6. Conception de l'espace de communication

Dans les deux chapitres précédents, nous avons présenté les aspects relatifs aux fonctions de production et de coordination de l'application Kiwi. Ces fonctions ont alors concerné des composants internes à l'espace de travail partagé du patch, mais aussi externes comme la

console, dans le cas de l'espace de production, ou encore l'interface de gestion des documents distants, dans le cas de l'espace de coordination. L'espace de communication, tel que nous l'avons défini en première partie de ce mémoire, comprend des fonctions permettant aux utilisateurs d'échanger des informations, et de se servir du système simplement comme d'un « messager » [Salber, 1995, p. 19]. Dans l'état actuel du logiciel, Kiwi ne dispose pas d'interfaces externes à l'espace de travail partagé, dédiées spécifiquement à la communication entre les personnes – comme on pourrait trouver dans des solutions collaboratives telles que *Google Docs* qui propose notamment une interface de discussion instantanée. D'autre part, une interface comme celle du *Beacon Dispatcher*[165] permet bien de transmettre des messages aux patch, mais uniquement de manière locale, tout comme le sont d'ailleurs les valeurs qui transitent à travers les différents objets au sein du patch lors de son exécution. Les fonctions de communication au sein de Kiwi se situent donc, dans la version actuelle de l'application, exclusivement au sein de l'espace interne du patch, et passent notamment via des objets tels que le *commentaire* ou encore l'objet *hub* que nous présenterons plus en détail ici.

Le simple fait d'avoir accès à un espace de travail partagé crée instantanément un potentiel d'interaction entre les participants à une session, qui s'emparent alors, presque automatiquement et de manière instinctive, de cet espace pour communiquer les uns avec les autres. Nous avons d'ailleurs pu observer ce phénomène à travers notre propre pratique. En effet, l'une des premières utilisations de Kiwi, dès que nous avons réussi à obtenir une version permettant une collaboration en temps réel, a été, non-pas de produire un traitement sonore complexe à plusieurs au sein d'un patch, mais de communiquer simplement entre nous en créant des objets, et en se satisfaisant de les voir apparaître sur l'espace de travail d'un autre en temps

[165] Cette interface est détaillée à la sous-section 4.4.2 (p. 157) de ce document.

réel. Ce type d'usage a ensuite pu être observé chez les étudiants découvrant le logiciel pour la première fois au sein de l'expérience pédagogique du cours pilote ayant eu lieu à l'Université. Ces derniers se sont tout de suite emparés de l'outil pour communiquer à distance avec les autres au sein de travail partagé du patch, notamment à l'aide d'objets *commentaire* interposés[166].

Les espaces définis par l'outil du trèfle des systèmes multi-utilisateurs ne sont pas figés. Comme nous le dit D. Salber, ils peuvent être dynamiques et évoluer au cours de l'interaction. Aussi, ces fonctions de communication peuvent très bien être au service d'une simple discussion entre les personnes, mais elles peuvent aussi servir à coordonner une activité comme celle de conception de traitement sonore ou encore servir d'une certaine manière à coordonner l'activité de jeu au sein du patch par la communication de données entre les participants. L'espace de production, tel que nous l'avons conçu et présenté au chapitre (4.), ne permet actuellement pas le partage des données d'exécution du patch entre les utilisateurs. Cet aspect de conception était pour nous essentiel afin de maintenir un contexte d'exécution local du patch. Néanmoins, le besoin de transmettre certaines informations entre les participants au sein d'un patch s'est fait ressentir, notamment en terme de pédagogie [Galleron & al., 2018]. Aussi avons-nous décidé en cours de projet d'y apporter une première réponse, qui s'est soldée par l'ajout d'un nouvel objet, nommé *hub*, à l'application.

Nous présenterons dans un premier temps cet objet en donnant quelques détails sur sa mise en œuvre, puis dans un second temps nous exposerons quelques cas d'utilisation. Nous verrons

[166] Nous aurons l'occasion d'évoquer ce point plus en détail dans la troisième partie de ce mémoire, lors de l'étude des retours liés à cette expérience pédagogique, à la sous-section 7.2.5 (p. 237) de ce document.

alors que les utilisateurs peuvent utiliser l'espace de travail partagé pour concevoir leurs propres canaux de communication à travers le patch en utilisant le potentiel du système.

6.1. Présentation et mise en œuvre de l'objet *hub*

L'objet *hub* a été créé en janvier 2018 et ajouté à la version *v1.0.0* de Kiwi de façon à pouvoir être testé et utilisé par les étudiants dans le cadre du cours pilote donné à l'Université Paris 8 avec Kiwi comme support logiciel [Galleron & al., 2018].

Lorsqu'il est utilisé dans le contexte d'un patch partagé, cet objet permet d'envoyer un message à tous les clients qui y sont connectés. La communication, à travers l'objet *hub*, se fait par l'intermédiaire de messages génériques. C'est-à-dire que les messages, reçus en entrée par l'objet sous la forme d'un *atom* ou d'une liste d'*atoms*, sont convertis en une chaîne de caractères pour pouvoir être transmis aux autres participants à travers le réseau. Lorsque l'objet *hub* est notifié d'un nouveau message, provenant par exemple d'un autre participant à la session, il convertit la chaîne de caractères en *atom* ou en liste d'*atom*s pour produire un message en sortie de l'objet.

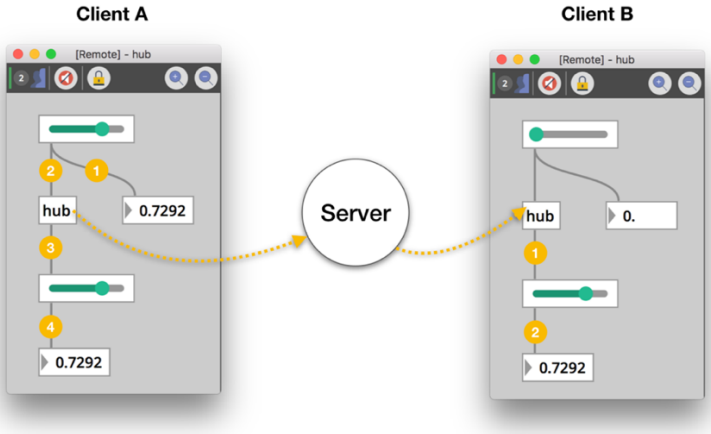

Figure 59 – Représentation simplifiée de la communication entre patchs permise par l'objet *hub*. Les nombres présents sur la figure correspondent à l'ordonnancement local des événements relatifs à chaque client.

La figure ci-dessus expose une représentation simplifiée de la communication permise à travers l'objet *hub* entre des répliques locales du même patch chez plusieurs utilisateurs connectés à la même session [Figure 59]. Dans le cas présenté, lorsque le client A modifie la valeur du potentiomètre linéaire (*slider*) sur l'interface graphique du patch, cela provoque l'envoi du message au sein de son propre patch à la *boîte-nombre* ainsi qu'à l'objet *hub*[167]. L'objet *hub* du client A génère alors l'envoi d'un message au sein du patch local aux différents objets connectés en sortie puis envoie le même message au serveur. Si un autre utilisateur est connecté au même document, comme c'est le cas ici pour le client B, l'information lui parvient aussi sous la forme d'un nouveau message envoyé en sortie de sa propre version locale de l'objet au sein du patch. La valeur des deux potentiomètres et des deux *boîtes-nombre* présentes en bas de chaque réplique locale du patch chez les différents clients connectés sont donc désormais mises à jour de manière parallèle. Le client A pourrait alors connecter celle-ci à d'autres objets pour mettre en place un contrôle collaboratif du patch. Néanmoins, le lecteur aura pu observer que seuls les objets connectés en sortie de l'objet *hub* sont synchronisés entre les utilisateurs. Le potentiomètre en haut du patch du client B ne correspond pas à la valeur du même potentiomètre chez le client A, ni à celui local à son patch en sortie de l'objet *hub*, dans la mesure où l'objet ne fait que *communiquer* une valeur à travers le réseau et ne permet pas nativement de synchroniser une même interface de contrôle au sein du patch.

En revanche, cette synchronisation peut être facilement mise en place par la programmation de cette fonctionnalité au sein du patch en bouclant la sortie de l'objet *hub* avec l'entrée d'une interface de contrôle. Ce bouclage ne peut néanmoins pas être direct, au risque de produire ce

[167] Notons que l'ordre dans lequel arriveront ces deux messages est ici indéterminé, comme nous l'avons spécifié dans le langage Patcher mis en œuvre dans Kiwi et présenté dans la sous-section 4.1.3 (p. 123) de ce document.

que l'on nomme un débordement de pile (*stack-overflow*), dû au fait que le même message serait envoyé à l'objet puis envoyé à nouveau de manière récursive, à l'infini.

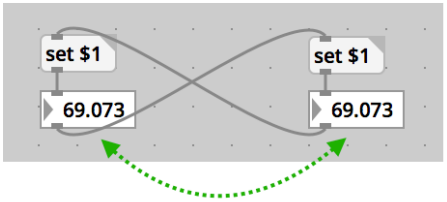

Figure 60 – Capture d'écran d'une partie de patch Kiwi dans laquelle est exposé un patron de conception permettant la synchronisation de deux interfaces graphiques, dans ce cas des *boîtes-nombres*, grâce à l'emploi de plusieurs messages comportant le texte « set $1 ».

Pour contourner ce problème, l'utilisateur pourra utiliser dans Kiwi le même patron de conception dont on se sert couramment dans les autres environnements de patching tels que Max ou Pure Data à savoir, faire précéder l'envoi d'un message à un objet graphique d'un message *set*, comme montré à la [Figure 60].

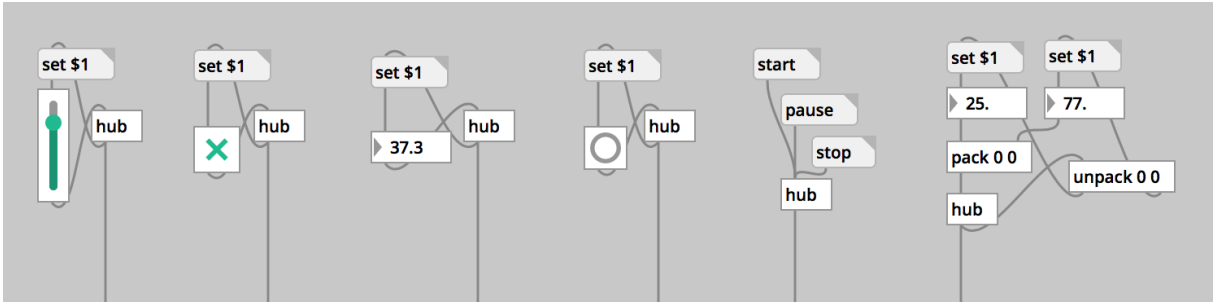

Figure 61 – Capture d'écran d'un patch Kiwi exposant un patron de conception de patch exploitant conjointement l'objet *hub* et une boîte-message comportant le texte *set $1* pour créer des interfaces de contrôle dont les valeurs sont partagées et synchronisées entre tous les participants connectés. Différents types de valeurs peuvent ainsi être transmises. De gauche à droite : un nombre flottant borné entre 0. et 1. à travers un potentiomètre linéaire (objet *slider*), une valeur booléenne (objet *toggle*), une valeur numérique non-bornée (objet *number*), un événement standard de type *bang*, des messages textuels ou encore des listes de plusieurs valeurs.

La méthode *set* a été généralisée à l'ensemble des objets graphiques de Kiwi depuis la version *v1.0.2* [Figure 61]. Ce message comporte dans Kiwi la même sémantique que dans les mises en œuvres du langage Patcher dans Max et Pure Data, à savoir, permettre la mise à jour de la valeur interne d'un objet, reflétée par l'interface graphique, en ne provoquant pas l'envoi d'un message en sortie. Les valeurs qui sortent de l'objet *hub* peuvent donc être utilisées à la fois pour contrôler directement des paramètres du patch mais aussi pour mettre à jour une interface de contrôle du patch pour ainsi renseigner l'ensemble des utilisateurs de sa valeur interne.

L'objet *hub* a été conçu autour de la classe *flip::Message* fourni par l'API de *flip* [Tableau 2, p. 128]. Le type *flip::Message* permet d'établir une communication isolée au sein d'un même document entre les personnes qui y sont connectées. Le fait d'avoir modélisé cet objet de cette manière implique que les notifications – les messages envoyés par l'intermédiaire de l'objet *hub* – ne sont pas persistantes. Cela a deux conséquences. La première est que les messages ne sont pas sauvés avec le document et donc que cet objet ne peut pas être utilisé pour le stockage d'une valeur au sein d'un patch. Lorsque l'on ferme et ouvre à nouveau un patch dont la valeur interne d'un objet a été synchronisée via l'objet *hub*, comme montré par exemple à la [Figure 61], la valeur interne de cet objet est réinitialisée à la valeur par défaut (en général 0.). La seconde conséquence est que lorsqu'un nouveau client se connecte à un document, les valeurs du patch communiquées à travers l'objet *hub* par les participants déjà présents sur la session ne sont pas présentes chez lui, l'information doit lui être renvoyée explicitement.

Notons que l'objet fonctionne aussi lorsque le patch est hors-ligne, c'est-à-dire non-connecté à un document distant ; dans ce cas, les messages reçus en entrée sont simplement renvoyés vers la sortie de l'objet. Son utilisation est donc possible aussi en local, néanmoins elle n'est

pertinente que dans le contexte où plusieurs personnes sont connectées en même temps à un même document distant.

En permettant la communication de données, l'objet *hub* ouvre la voie à de nouvelles possibilités d'interaction avec le patch qui relèvent désormais du jeu et non-plus de la simple édition de document. Nous donnons dans la section suivante quelques exemples possibles d'utilisation de l'objet *hub* au sein d'un patch.

6.2. Utilisations de l'objet *hub*

Le premier exemple d'utilisation, qui sans doute le plus commun, est de se servir de cet objet pour synchroniser des paramètres de jeu au sein du patch. Le second que nous donnerons permet de fabriquer un mécanisme élémentaire de discussion instantanée inter-utilisateur en se servant des différents composants de l'interface de l'application. Le troisième permet d'obtenir une information au sein du patch, telle que le nombre d'utilisateurs actuellement connectés à une session, par la conception d'un système simple de requête inter-utilisateur.

6.2.1. Synchronisation de valeurs de jeu

L'objet hub permet, comme nous l'avons vu précédemment, la communication de valeurs de jeu. En combinant cet objet avec un objet *message* permettant de préfixer l'envoi d'une valeur du texte « set », il est possible d'orchestrer une synchronisation de ces valeurs chez les participants connectés à une session afin qu'ils disposent tous des mêmes valeurs de contrôle du patch.

La figure ci-dessous expose une version mise à jour du patch présenté initialement à la [Figure 31, p. 151] dans le cadre de la sous-section 4.3.5. (p. 150) de ce document, dédiée au contexte d'exécution local du patch.

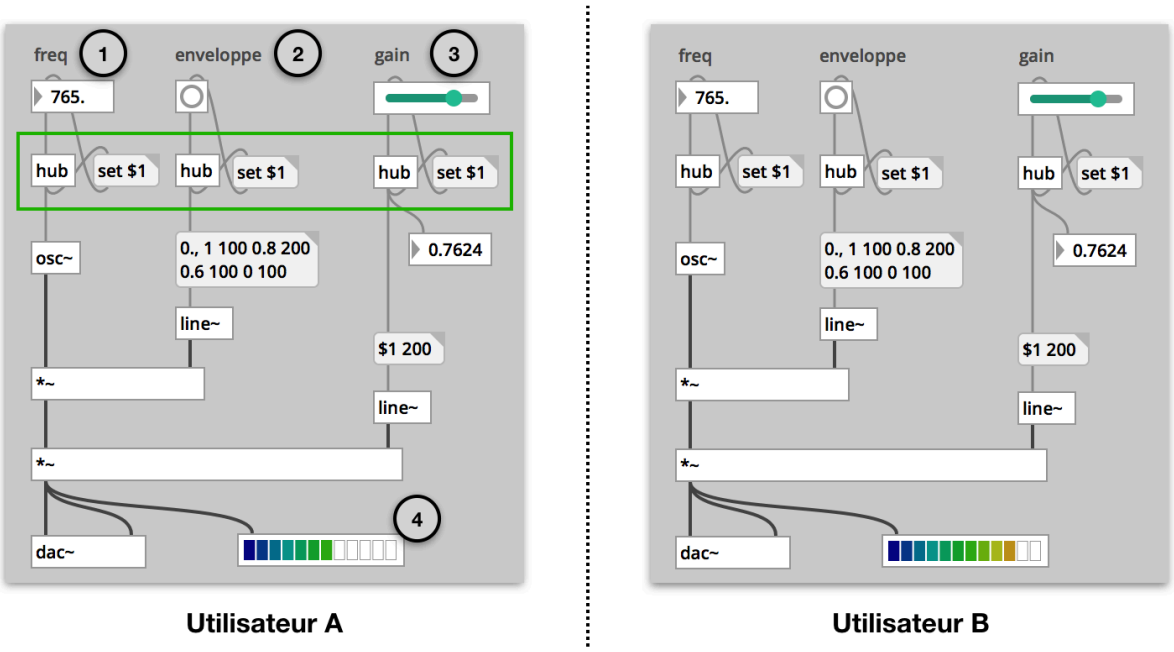

Figure 62 – Capture d'écran d'un même patch Kiwi tel qu'il apparaît chez deux utilisateurs différents, connectés en temps réel au même document. Les numéros présents sur le patch de l'utilisateur A sont repris dans le corps du texte qui suit. Nous nous réfèrerons par exemple à l'objet *meter~* en utilisant le code (4). Le rectangle vert correspond aux modifications apportées ici par rapport à la version du même patch présenté précédemment à la [Figure 31, p. 151].

Le patch présent ici a été mis à jour de manière à ce que les trois paramètres du patch soient synchronisés chez l'ensemble des participants à la session [Figure 62]. Pour cela, le patron de conception permettant la synchronisation de ces interfaces de contrôle, présenté un peu plus haut à la [Figure 62, p. 220], a été utilisé. On pourra observer que les deux utilisateurs disposent bien des même valeurs internes de fréquence [Figure 62, (1)] et de gain [Figure 62, (3)]. À chaque fois que l'un des deux modifie la valeur d'une de ces interfaces, cette valeur est transmise à tous les autres utilisateurs. De la même façon, lorsqu'un utilisateur déclenche la production d'une enveloppe d'amplitude [Figure 62, (2)], la production de cette enveloppe est aussi déclenchée chez les autres.

Si les différentes interfaces de contrôle sont synchronisées entre-elles, ce n'est pas le cas de l'interface de visualisation du *meter~* [Figure 62, (2)] du fait que le traitement audio, qui s'exécute localement, n'est en fait pas exactement le même chez les deux participants. Le message envoyé par un utilisateur à travers l'objet *hub* est automatiquement envoyé localement en sortie, garantissant ainsi un temps de réponse minimal du système. En revanche, le temps de notification est beaucoup plus long[168]. En effet, l'emploi du *hub* implique forcement une latence entre l'envoi d'un message par un utilisateur et sa réception chez un autre utilisateur dans la mesure où le message doit transiter par le serveur avant de lui être transmis. Ceci explique alors que l'audio soit légèrement retardé chez l'utilisateur B par rapport à l'utilisateur A, du fait qu'il ait déclenché la production d'une nouvelle enveloppe d'amplitude en appuyant sur le bouton prévu à cet effet au sein du patch mais que cette information n'ait été transmise qu'après un certain temps à l'autre utilisateur. Notons que si ce comportement peut être acceptable si les participants sont géographiquement dispersés, il peut donc se réveler plus gênant lorsqu'ils sont colocalisés.

6.2.2. Création d'un mécanisme de discussion instantanée

La solution *netpd*, étudiée en première partie de ce mémoire de thèse[169], offre une interface sommaire de *Chat* sous la forme d'un patch Pure Data permettant une discussion instantanée entre les utilisateurs d'une session de jeu en réseau [Figure 15, p. 81, en haut à gauche]. Comme nous l'avons dit, Kiwi ne dispose pour l'instant pas de composant spécifiquement dédié à cela dans l'interface qu'elle propose nativement aux utilisateurs. Néanmoins, il est intéressant de

[168] Les notions de temps de réponse et de temps de notification sont définies à la section 3.1 (p. 88) de ce document.
[169] Voir 2.2.2.2 (p. 80) de ce document.

noter qu'il est possible pour un utilisateur de *fabriquer* directement un système s'en rapprochant, grâce à l'emploi de l'objet *hub*[170].

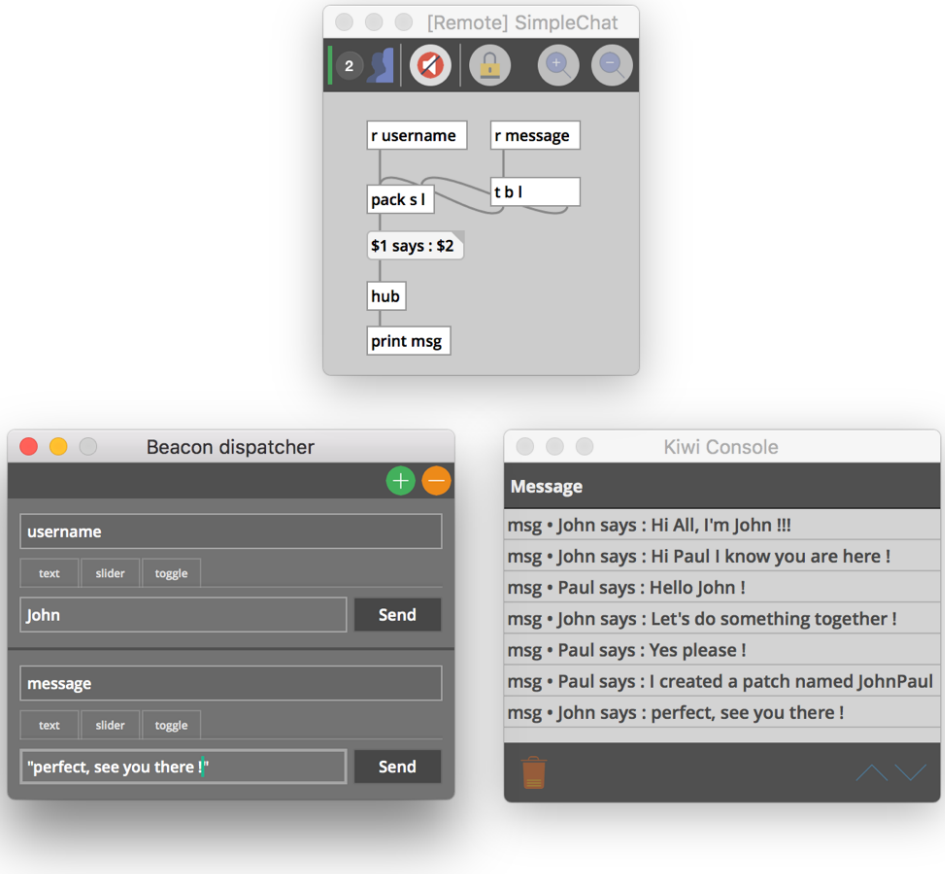

Figure 63 – Capture d'écran du logiciel Kiwi exposant un mécanisme de discussion instantanée fabriqué à l'aide d'un patch, visualisé dans l'interface de Console et contrôlé par la fenêtre *Beacon Dispatcher*.

La figure ci-dessus expose trois composants de l'interface graphique qui, combinés entre eux, peuvent servir à créer un mécanisme de discussion instantanée entre les utilisateurs [Figure 63]. L'interface de *Beacon Dispatcher*[171] est utilisée pour envoyer des messages aux objets *receive*

[170] Notons que la solution *bricolée* que nous exposons ici ne vise pas à répondre pas de manière convenable à la problématique de la communication inter-utilisateur au sein du logiciel. Elle a plus un intérêt didactique visant à exposer un cas d'utilisation original de l'objet *hub*.

[171] Cette interface est décrite en détail à la sous-section 4.4.2 (p. 157) de ce document.

du patch intitulé *SimpleChat*. Chaque participant est d'abord invité à renseigner son nom d'utilisateur au sein du champ *username* du *Beacon Dispatcher*. Lorsqu'un des utilisateurs tape et envoie un message au sein de cette même interface, celui-ci est transmis à l'objet *receive* portant le *symbole* « message » qui, associé au nom de l'utilisateur local renseigné au préalable, est formaté à l'aide de l'objet *message*, puis transmis à l'objet *hub* qui le transmet à son tour localement à l'objet *print* mais aussi aux autres participants à la session à travers l'objet *hub*. Chaque participant à la session peut alors suivre la discussion en temps réel sur sa console[172] personnelle qui affiche l'historique complet de la conversation.

6.2.3. Obtention du nombre d'utilisateurs connectés au sein du patch

Pouvoir connaître le nombre d'utilisateurs actuellement connectés à un patch est important dans le cadre d'une application collaborative dans la mesure où il améliore la *conscience de groupe*. Cette information est visible dans l'interface graphique, en haut à gauche au sein de la barre d'outil du patch [Figure 38, p. 165]. En revanche, aucun objet ne permet pour l'instant de la récupérer facilement au sein d'un patch. Or cette information pourrait très bien être exploitée à des fins musicales en tant que variable d'exécution, notamment pour contrôler un traitement sonore de façon conditionnelle au nombre d'utilisateurs présents. On peut imaginer un cas d'utilisation très simple qui consisterait à programmer une baisse du niveau sonore général d'un patch de trois décibels par nombre de personnes connectées afin que, dans un contexte où tous les participants sont colocalisés et exécutent tous une instance différente d'un même patch sur leur machine, imaginons dans le cadre d'une pratique de *Laptop Orchestras* [Dannenberg & al., 2007] ou encore de *live coding* [Haefeli, 2013], le niveau sonore reste à peu près constant dans la salle.

---

[172] Ce composant est décrit en détail à la sous-section 4.4.1 (p. 153) de ce document.

Pour ce faire, une possibilité serait d'utiliser l'objet *hub*. Cet objet peut en effet servir, comme on l'a vu aussi dans la sous-section précédente, à d'autres fins que du simple contrôle de paramètres de jeu. On peut l'utiliser ici comme brique élémentaire à la construction d'un mécanisme permettant d'effectuer des requêtes, c'est-à-dire un mécanisme de questions-réponses inter-utilisateurs, afin de connaître par exemple le nombre de participants actuels d'une session au sein d'un patch.

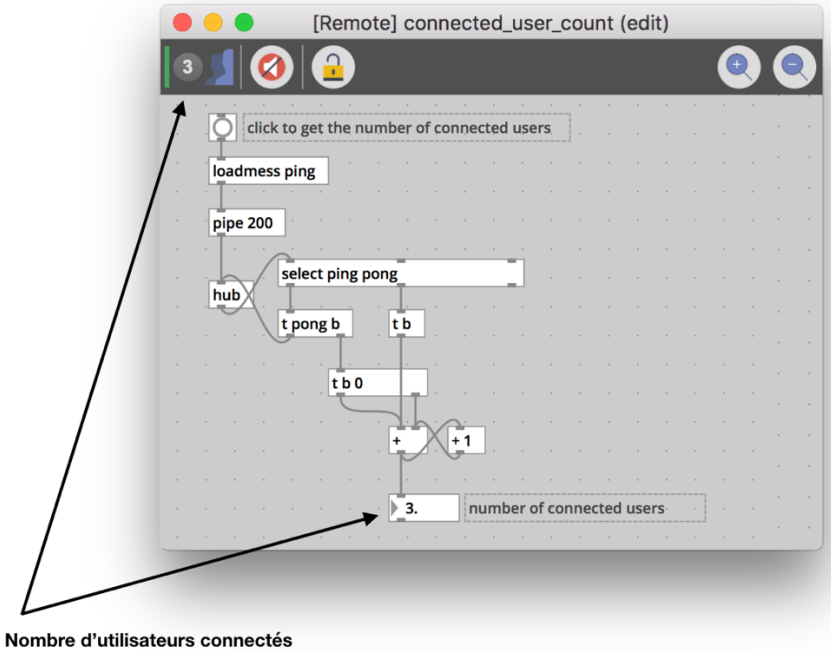

Figure 64 – Capture d'écran d'un patch Kiwi montrant une technique de programmation utilisant l'objet *hub* pour obtenir le nombre d'utilisateurs actuellement présents/connectés sur le patch.

La figure ci-dessus expose une première mise en œuvre sous forme de patch dont le fonctionnement logique est le suivant : une requête est envoyée à tous les utilisateurs via le message *ping* à travers l'objet *hub*. La réception de ce message chez chacun des utilisateurs provoque une remise à zéro de leur compteur local, puis déclenche l'envoi d'une réponse envoyée à tous les utilisateurs via le message *pong*, à travers leur propre instance du même

objet *hub*. A chaque message *pong* reçu, le compteur local de chaque utilisateur est incrémenté. Le nombre de personnes actuellement connectées au patch est alors égal au nombre total de messages *pong* reçus [Figure 64]. Cette variable, injectée au sein de l'espace de jeu du patch peut alors servir à contrôler toute sorte d'interaction de manière conditionnelle.

Dans le système présenté plus haut, le nombre d'utilisateurs est automatiquement mis à jour chez tous les clients, dès que quelqu'un appuie sur le bouton principal du patch ou se connecte au document distant grâce à l'objet *loadmess*. Il reste néanmoins perfectible dans la mesure où il peut subir des problèmes de concurrence – lorsqu'un utilisateur envoie un nouveau *ping* alors que tous les messages *pong* n'ont pas encore été reçus par exemple – mais aussi car l'information n'est pas mise à jour au moment où un utilisateur quitte le patch. Il faudrait pour cela ajouter à Kiwi un objet qui notifie le patch par un message, envoyé juste avant la fermeture de celui-ci, similaire par exemple dans son fonctionnement à l'objet *closebang* que l'on peut trouver dans l'environnement Max. Cette méthode souffre donc de quelques problèmes inhérents à sa mise en place sous forme de patch, et nous n'excluons pas de rajouter cette fonctionnalité sous la forme d'un nouvel objet natif autonome, ou au sein d'un objet plus général, dans une prochaine version de l'application. Il représente néanmoins un cas d'utilisation original de l'objet *hub* qui ouvre de nombreuses perspectives encore à découvrir.

Nous avons exposé, dans la partie centrale de ce mémoire, l'application Kiwi telle qu'elle se présente à l'utilisateur dans sa version actuelle. Nous avons alors étudié les aspects relatifs à la conception des espaces de production, de coordination et de communication en suivant la trame tripartite offerte par l'outil conceptuel du trèfle des systèmes multi-utilisateurs. Cette application est à considérer en l'état, mais aussi comme un travail encore en progression, dans la mesure où le projet MUSICOLL court jusqu'en décembre 2018 et que de nombreuses

fonctionnalités sont encore absentes ou demanderaient à être améliorées. Néanmoins, elle a déjà pu être utilisée à de nombreuses reprises lors de diverses représentations musicales et dans différents contextes pédagogiques. La dernière partie de ce mémoire se consacrera donc à l'analyse des retours que nous avons pu obtenir mais aussi à l'exposé des différentes perspectives du projet qui restent encore ouvertes.

Partie III – Retours d'utilisation et perspectives

Nous reviendrons dans un premier temps sur différents cas d'utilisation du logiciel Kiwi rencontrés durant le projet MUSICOLL. Ils nous permettront d'évaluer la solution proposée sur le plan de la pratique musicale et de la pédagogie. Nous verrons alors ce qu'apporte Kiwi à ces différents cas d'usage mais aussi certaines limites à l'application telle qu'elle se présente actuellement. Ceci nous amènera à envisager des perspectives futures pour le projet, notamment pour améliorer le support des différents types d'interaction. Certaines des fonctionnalités que nous allons envisager sont par ailleurs déjà en cours de développement alors que d'autres sont à considérer à plus long terme. Nous ouvrirons donc quelques pistes de recherche et donnerons des solutions qui nous paraissent envisageables à moyen ou plus long terme.

7. Premières utilisations du logiciel

L'application Kiwi est libre, gratuite et accessible en ligne. Elle a été téléchargée plus de 570 fois, compte aujourd'hui plus de 75 utilisateurs inscrits et environ 200 documents hébergés en ligne ont été créés depuis le début du projet MUSICOLL[173]. Malgré cela, nous avons encore bénéficié de peu de retours spontanés de la part d'utilisateurs extérieurs au projet. En revanche, l'équipe a pu s'en servir à de nombreuses reprises, notamment lors de réunions hebdomadaires où nous étions réunis dans la même salle autour du même patch sur différentes machines, ou encore lors de sessions de travail distribuées géographiquement, alors souvent accompagnées d'un support de visio-conférence. D'autre part, l'application a été présentée par l'équipe à diverses occasions, au sein de conférences, de démonstrations en France et à l'étranger, et

[173] Ces chiffres correspondent à une mesure effectuée le 13 août 2018.

utilisée dans le cadre d'ateliers, de concerts ou encore comme support de cours à l'Université. Il est aussi prévu qu'elle soit intégrée au sein d'un MOOC[174] à partir de novembre 2018. Nous revenons sur ces différents cas d'utilisation ici afin de mieux cibler le type de pratiques auxquelles le logiciel Kiwi se propose de répondre.

7.1. Festival la DéMO

Dans le cadre de l'édition 2017 du festival *La DéMO*, organisé par le conservatoire de musique et de danse de Saint-Denis, l'équipe MUSICOLL a pu animer un atelier de « patching collaboratif » avec des collégiens à la médiathèque *Don Quichotte* de Saint-Denis.

Figure 65 - Photographies prises dans le cadre du festival *La DéMO* organisé à Saint-Denis en 2017. À gauche, on peut voir des étudiants du conservatoire en train de réaliser des patchs de traitement et de synthèse du son par petits groupes, à la médiathèque *Don Quichotte* de Saint-Denis. A droite, un second groupe d'élèves manipulant un instrument physique traité en temps réel par Kiwi lors d'une représentation publique à l'auditorium de la MSH Paris-Nord.

Une dizaine de collégiens étaient alors réunis par petits groupes de deux ou trois, sur quatre machines différentes fournies par la médiathèque, pour cette expérience qui consistait à réaliser

[174] Acronyme de langue anglaise *Massive Open Online Course* désignant une formation à distance en ligne ouverte à tous.

des patchs de traitement et de synthèse sonore de manière collaborative [Figure 65, à gauche]. Parallèlement, un autre groupe d'élève s'est consacré à la fabrication d'instruments de musique, physiques cette fois-ci, réalisés à partir de matériau de récupération[175]. Les réalisations respectives de ces deux groupes ont ensuite fusionné lors d'une représentation musicale publique et collective donnée au sein de l'auditorium de la MSH Paris-Nord le jour même [Figure 65, à droite]. Les luthiers numériques en herbe traitaient, à travers les patchs qu'ils avaient auparavant réalisés avec Kiwi, les instruments acoustiques créés parallèlement par l'autre groupe d'élèves, en temps réel, dans la tradition de la musique mixte.

Dans le cadre de cette représentation publique, nous avons décidé de projeter les différents patchs des étudiants sur un écran en fond de scène afin que le public puisse les voir évoluer et ainsi faciliter la vulgarisation de l'ensemble de la démarche pédagogique et musicale auprès du public [Figure 65, à droite]. Selon Zmölnig, « la métaphore simple du flux de données permet aux personnes non-familière avec le langage d'avoir au moins une idée générale de ce qui est en train de se passer »[176]. Même s'il ne s'agissait ici que d'exposer la variation des différents paramètres de contrôle du patch par les élèves, ce choix a été fait un peu dans le même esprit que dans le cadre des pratiques de *live coding*, telles que décrites notamment dans [Zmölnig, 2007] ou [Pd~Graz, 2009], qui projetaient de la même manière le patch en train d'être édité à plusieurs, sur scène.

Cette expérience représente pour nous un excellent exemple de la séparation des activités de conception et d'exécution qui peuvent exister dans les environnements de patching. Dans ce

---

[175] Cet atelier a été animé par Nicolas Bras : http://www.musiquesdenullepart.com/ [consulté le 18/09/18].

[176] [Zmölnig, 2007], p. 1., et traduit de l'anglais : « the simple metaphor of dataflow allows people who are not familiar with the language to at least get an overall idea what might be going on. ».

cas, les deux ateliers étaient constitués de deux groupes différents. Le groupe créant les traitements ne savaient pas exactement quelles seraient les données audio envoyées au patch, de l'autre côté, les élèves construisant les instruments physiques ignoraient complètement la manière dont ils allaient être traités en temps réel lors de la représentation. Il a donc fallu un effort d'abstraction et de généralisation de la part des concepteurs de patch pour anticiper le son que pourrait produire leur patch lors du concert. Nous avons pu dans ce cadre apprécier le choix qui a été fait de pouvoir écouter le rendu d'un patch localement sans que cela ne vienne affecter le travail réalisé par quelqu'un d'autre simultanément sur le même patch. Chacun de ces petits groupes a créé un patch sur le serveur en ligne, destiné à être joué de manière indépendante lors du concert. Chaque mini-groupe pouvait alors *observer* ou interagir de manière inédite avec les autres, en ouvrant simplement les documents que les autres groupes avaient créés pour les comparer au leur, et écouter le rendu sonore sans affecter le rendu local de ce même patch, exécuté aussi sur les autres machines.

Ces collégiens furent en fait les tout premiers utilisateurs du logiciel Kiwi. Aussi, les premiers retours que nous avons obtenus de leurs parts étaient donc très important pour nous dans le cadre du projet MUSICOLL. Cette manifestation a pu prouver notamment l'intérêt de l'accessibilité des ressources établies dans les spécifications initiales du logiciel. Le stockage des patchs sur le serveur et leur facilité d'accès à partir du logiciel a permis une migration transparente des patchs réalisés à la médiathèque, vers l'ordinateur dédié à l'exécution dans l'auditorium. Les patchs créés sur le serveur distant, et sur lesquels les élèves avaient travaillés à la médiathèque ont simplement pu être rouverts une fois arrivés dans l'auditorium sur la machine dédiée au concert, pour y être utilisés.

Au moment où s'est tenu cet événement, Kiwi ne comportait pas encore d'interface graphique au sein du patch, son contrôle passait alors exclusivement par le composant graphique *Beacon Dispatcher*[177], depuis une fenêtre extérieure au patch. Cette manifestation nous a donc fait prendre conscience que la solution provisoire du contrôle externe au patch était loin d'être satisfaisante d'un point de vue de l'expérience utilisateur, et nous a donc poussé à accélérer la mise en œuvre des objets graphiques au sein de Kiwi, ceci pour faciliter et permettre le contrôle directement au sein du patch.

Cette expérience a montré selon nous aussi, de manière générale, l'engouement des collégiens pour la pratique du temps réel, leur prise en main relativement facile du logiciel, et l'intérêt pédagogique et ludique de l'outil. Précisons à ce propos que les collégiens commencent au même moment l'étude de la physique en classe, et que Kiwi et la synthèse temps réel leur donnent une expérience sensible immédiate de concepts qui peuvent être abstraits. Si certains étaient aux premiers abords un peu déroutés par cette nouvelle approche qu'ils découvraient, d'autres en revanche s'y sont très vite fait et sont même retournés sur les logiciels que nous avions laissés à leur disposition sur les différents postes de la médiathèque suite à cet atelier pour continuer à expérimenter.

7.2. Cours pilote avec Kiwi à l'Université Paris 8

Outre la production d'une maquette d'environnement de patching audio collaboratif, le second objectif affiché du projet MUSICOLL était d'étudier le renouvellement qu'il pourrait induire dans la pédagogie musicale. Si une première expérience avait pu être menée avec des collégiens dans le cadre du festival *La DéMO*, évoqué précédemment, nous voulions aussi étudier les

[177] Cette fenêtre est décrite en détail dans la seconde partie de ce document, consacrée à la mise en œuvre du logiciel Kiwi. Voir partie 4.4.2 (p. 157).

enjeux liés à son utilisation dans le cadre de l'enseignement supérieur dédié à la pratique de la composition musicale.

7.2.1. Contexte

Il a donc été décidé en début de projet de refondre avec Kiwi le cours universitaire de Licence 2 d'introduction aux langages de patching audio donné en présentiel, initialement avec Max et Pure Data[178]. Si cet objectif n'était pas le seul enjeu du projet et de la création de Kiwi, celui-ci a énormément guidé et orienté son développement, comme nous avons déjà eu l'occasion de l'évoquer au cours de ce mémoire. De manière réciproque, les nouvelles potentialités d'interaction offertes par l'outil ont permis de repenser et d'adapter de manière dynamique le contenu pédagogique du cours et la manière de l'enseigner. Cette refonte du cours et ses enjeux – qui étaient principalement de « renforcer […] le *savoir-apprendre* et le *savoir-travailler* de manière collaborative afin de montrer aux étudiants une manière alternative d'apprendre, de travailler et de créer »[179] – ont été développés en détails dans un article d'équipe présenté au Journées d'Informatique Musicale en 2018, aussi nous renvoyons le lecteur vers ce texte [Galleron & al., 2018].

7.2.2. Premiers tests

Le cours en présentiel a été précédé d'une séance de groupes nommée *crashtest*, en décembre 2017, au sein de laquelle les étudiants étaient invités à prendre en main le logiciel pour la première fois. Cette séance a été initialement conçue pour effectuer des tests en situation réelle auprès des étudiants et nous permettre d'obtenir des premiers retours sur le fonctionnement

[178] Dans le cadre de la mineure CAO (Composition Assistée par Ordinateur) au sein du département de Musique de l'Université Paris 8

[179] [Galleron & al., 2018, p. 108]

général de l'application. Elle nous a permis, en tant que développeurs, de faire émerger les principaux problèmes – notamment liés aux conflits générés par une utilisation importante de la collaboration, en testant une montée en charge du système avec 21 étudiants connectés sur le même patch. Nous avons alors corrigé ces dysfonctionnements avant la mise en production de la première version *beta* de Kiwi utilisable (*v1.0.0*), qui était alors prévue vers fin janvier 2018. Au cours de cette séance, l'équipe a pu aussi observer et analyser les interactions entre les utilisateurs se saisissant de l'outil pour la première fois et tirer des premières conclusions sur les pistes de recherche à court terme devant être menées avant la mise en place du cours. Les étudiants se sont saisis assez rapidement de l'outil en l'appréhendant de manière assez ludique par la création d'objets ou de commentaires qu'ils voyaient alors apparaître automatiquement sur la machine de leurs pairs. En revanche, certains se sont étonnés, en bougeant par exemple la valeur d'un potentiomètre, que cette action n'ait pas d'impact chez les autres. Aussi a-t-il été décidé à ce moment d'apporter une première réponse à la dimension du jeu collaboratif en développant l'objet *hub*, que nous avons présenté dans la seconde partie de ce mémoire[180]. Cet objet a d'ailleurs représenté une porte d'entrée pédagogique lors de la première séance du cours, donné par E. Maestri, afin d'introduire les étudiants à la dimension collaborative, avant d'aborder des concepts plus techniques liés à l'algorithmique ou au traitement du signal.

7.2.3. Reconfiguration des usages

Traditionnellement, pour montrer son travail à la classe ou au professeur, les étudiants devaient soit le faire à partir de leur propre machine, en la reliant aux équipements de la salle (projection vidéo / sonorisation), soit copier le patch sur la machine centrale de la classe, celle-ci étant généralement déjà reliée au système. La première solution était souvent chronophage, et posait

[180] Voir section 6.1. (p. 215) de ce document.

par exemple certains problèmes de compatibilité (adaptateurs manquants, drivers audio non-installés, etc). La seconde n'était pas non-plus toujours satisfaisante dans la mesure où, si le patch était édité sur la machine centrale – notamment pour corriger un problème ou mettre en place de nouvelles idées à partir du travail fourni par l'étudiant – celui-ci se retrouvait avec une nouvelle version du patch à devoir rapatrier sur sa machine, apportant alors une confusion sur la version à devoir gérer. Le logiciel Kiwi, tel qu'il a été conçu, a permis une reconfiguration des usages traditionnels en matière de pédagogie à différents niveaux. Dans un premier temps, il a permis d'améliorer le flux de travail (*workflow*) dans le cadre du cours en permettant aux différents acteurs le stockage et l'accès aux patchs directement en ligne. Ce qui a résolu alors notamment les problèmes liés au transfert de documents évoqués plus haut. Il a permis aussi aux personnes de retrouver leur travail depuis n'importe quel poste et ouvre donc la voie à une collaboration asynchrone sur les différents patchs. Ce type de collaboration a permis notamment d'échanger plus facilement une information avec un pair ou avec le professeur, améliorant ainsi la co-résolution de problèmes. D'autre part, le fait de pouvoir se connecter aux mêmes documents, en temps réel ou en temps différé, autorise des interactions symétriques entre les étudiants du même niveau, mais aussi asymétrique, c'est-à-dire entre des participants avancés et moins avancés (par exemple entre des étudiants de différents niveaux ou entre un professeur et des étudiants) qui « communiquent des stratégies de partage d'objets numériques et renforcent ainsi l'apprentissage réciproque dans et à travers leur utilisation » [Galleron & al., 108]. Cette dimension a pu être observé à la fois lors du cours en présentiel mais est visible, comme nous l'observerons un peu plus loin aussi sur les documents créés par les étudiants qui en laissent encore des traces.

7.2.4. De nouvelles interactions

Le schéma ci-dessous représente l'espace physique de la classe, qui a été reconfiguré durant les cours en présentiel, et les interactions possibles des différents acteurs du cours à travers l'accès aux documents partagés sur leurs propres machines, à travers l'application Kiwi [Figure 66].

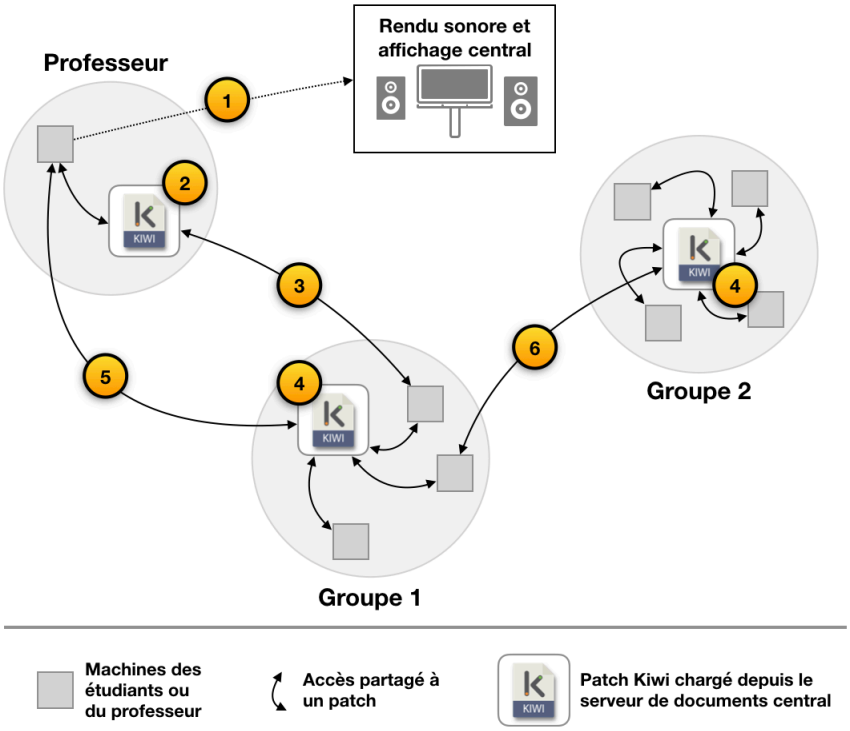

Figure 66 – Schéma simplifié, illustrant la redéfinition de l'espace de la classe durant l'expérience du cours pilote avec Kiwi comme support logiciel lors d'une séance en présentiel. Les étudiants sont réunis par petits groupes autour d'un même patch commun auquel ils ont accès via le serveur de documents centralisé de Kiwi. Les flèches montrent ici les interactions possibles entre les différents acteurs du cours au niveau de ces documents. Nous nous réfèrerons dans le corps du texte qui suit aux différents numéros présents sur cette figure en utilisant par exemple le code (1) pour signifier le lien entre la machine du professeur et le système d'affichage et de rendu sonore central.

La machine du professeur est reliée à un système central d'affichage et de sortie audio permettant d'exposer un traitement sonore à l'ensemble de la salle (1). Le patch que le professeur expose peut, comme dans le cadre d'un cours traditionnel du même type, être local, c'est-à-dire chargé depuis sa propre machine. Mais il peut aussi correspondre à un document distant, chargé depuis le serveur central de Kiwi (2). Dans ce dernier cas, chaque étudiant peut

alors le visualiser directement sur sa propre machine et y interagir en temps réel (3), sans avoir à copier manuellement son contenu en reproduisant ce que fait le professeur. Pour cela, il lui suffit de se connecter lui aussi au même document depuis son application. L'étudiant peut alors, sans gêner les autres, tester le rendu sonore du patch localement, en utilisant par exemple un casque, dans la mesure où le contexte d'exécution du patch lui appartient. Si l'étudiant a une question, il peut alors simplement pointer un objet, un lien ou un groupe d'éléments en effectuant une sélection sur sa version locale du patch pour appuyer son propos, en rendant automatiquement cette sélection visible sur le patch partagé au sein de l'espace d'affichage central (mais aussi sur l'ensemble des machines des étudiants s'y étant connecté). Pour favoriser un apprentissage collectif et collaboratif, et permettre aux étudiants de se « former à et par la collaboration numérique » [De Lavergne & Heïd, 2013], l'équipe pédagogique a décidé de créer trois groupes constitués de cinq à sept étudiants travaillant ensemble à la création d'un même traitement sonore à plusieurs [Figure 66]. Le but final étant de faire réaliser aux trois groupes une pièce musicale qu'ils devraient interpréter de manière collaborative lors d'un concert public lors de la dernière séance en guise d'évaluation. Chaque groupe a alors créé un document commun sur le serveur central auquel chaque participant du groupe était connecté (4). Lorsqu'un étudiant ou un groupe avait une question spécifique sur un patch, le professeur pouvait intervenir directement en se connectant au document (5). Il pouvait aussi exposer le problème à toute la classe en le projetant à l'écran si cela se révélait pédagogiquement pertinent (1). De la même façon, une résolution de problème entre pairs était aussi possible, les étudiants pouvant se connecter en temps réel aux documents en train d'être édités par un autre groupe (6).

Si on reprend l'outil conceptuel fourni par la matrice d'Ellis, exposée en première partie de ce mémoire [Tableau 1, p. 29], nous pouvons dire que Kiwi s'est alors adapté ici à une situation

pédagogique où l'interaction s'effectuait en temps réel, depuis plusieurs machines mais de manière colocalisée au sein d'une même salle de classe. Mais l'application a aussi servi dans le cadre de ce cours à d'autres types d'interaction. En effet, les élèves rentrant chez eux pouvaient aussi avancer sur leur composition, en contribuant chacun à leur rythme sur le document de manière asynchrone. Les étudiants ont aussi travaillé de manière géographiquement distribuée, depuis leur propre machine, lors de sessions de travail entre pairs en temps réel, ou lors de sessions de travail codirigées avec le professeur, assistées notamment d'un support de vidéo-conférence.

7.2.5. Communication inter-utilisateur

Grâce au caractère ouvert de l'application Kiwi, et le libre accès laissé aux documents distants par l'intermédiaire de la fenêtre *Document Browser*, nous avons pu observer durant tout le semestre l'élaboration progressive des patchs par les étudiants en ouvrant directement les documents sur lesquels ils travaillaient. Comme nous l'avons dit, Kiwi n'offre aucune interface dédiée spécifiquement à la communication inter-utilisateur telle qu'une interface de discussion instantanée ; ni de composants spécifiques dédiés à la coordination des tâches tels que ceux proposés par K. Touskalas, comme un « Post-it » qui permettrait aux utilisateurs de laisser des notes explicatives aux autres utilisateurs ou d'un composant de type « ToDo » qui permettrait d'apporter des informations sur les orientations futures d'un projet ou d'un patch en particulier [Touskalas, 2011]. Néanmoins, en étudiant les différents patchs produits par les étudiants, nous avons pu constater qu'ils se sont eux-mêmes créés leurs propres systèmes de communication au sein du logiciel à travers l'espace commun du patch, en utilisant les moyens simples qui étaient à leur disposition, et notamment l'objet *commentaire*. La figure ci-dessous en expose une sélection [Figure 67].

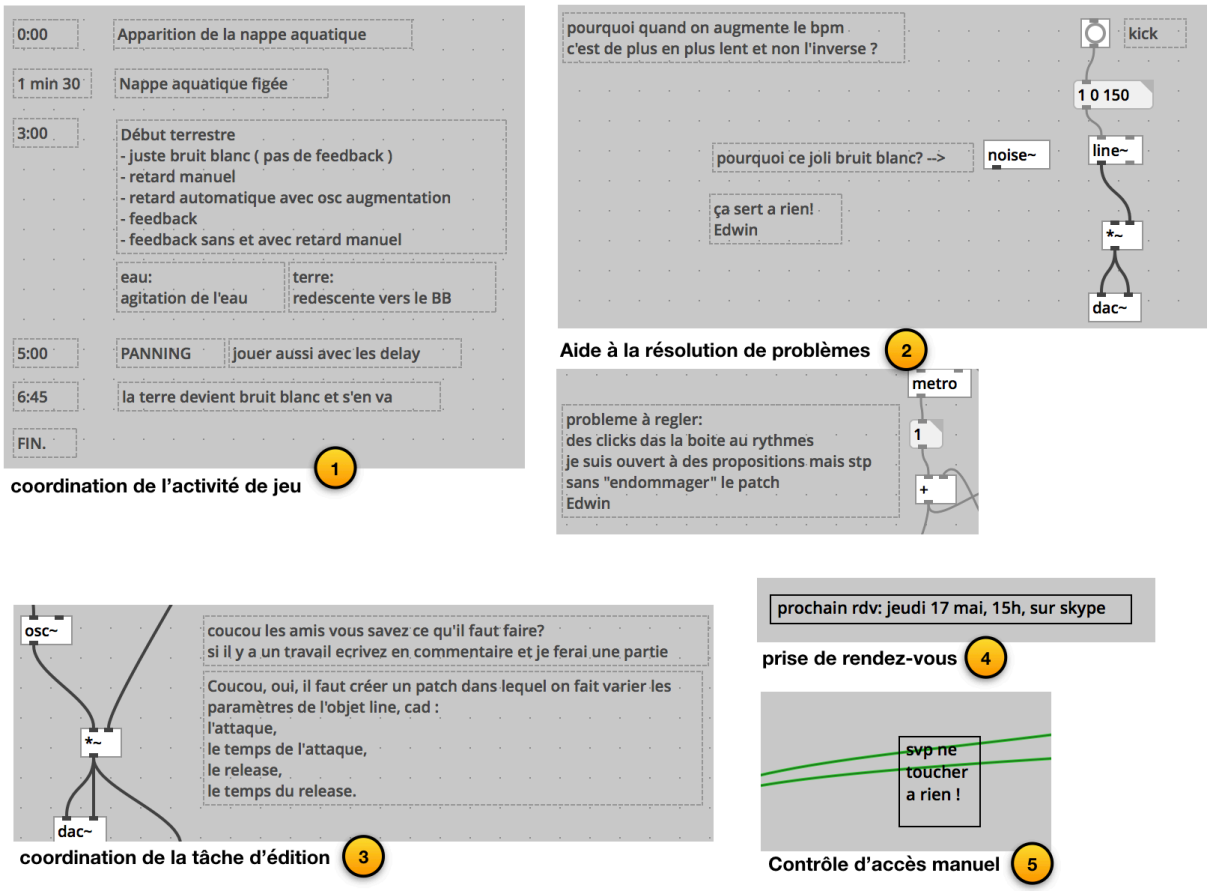

Figure 67 – Sélection de quelques commentaires laissés par les étudiants au sein de différents patchs qu'ils ont créés durant le cours pilote donné avec Kiwi comme support. Nous nous réfèrerons dans le corps du texte qui suit aux différents numéros présents sur cette figure en utilisant par exemple le code (2) pour désigner les commentaires relatifs à « l'aide à la résolution de problèmes ». Notons que les captures (4) et (5) ont été prises sur des patchs issus de la version *v1.0.0* du logiciel.

Dans la mesure où le patch devient un espace commun qu'ils peuvent venir consulter en même temps ou de manière asynchrone, les étudiants ont utilisé cet objet *commentaire* pour communiquer entre eux lorsqu'ils étaient connectés au même moment, notamment afin de coordonner leurs activités en se fixant par exemple des rendez-vous (4) ; coordonner l'activité d'édition en posant les bases du travail qui reste à effectuer sur un patch (3) ; poser des questions et les résoudre à plusieurs comme on pourrait le faire sur un forum en ligne (2) ; contrôler l'accès au document de façon manuelle – dans la mesure où la solution logicielle ne le permet pas encore (5) ; ou encore coordonner l'activité de jeu collaboratif en temps réel (1), notamment

à l'occasion de la représentation musicale finale à la dernière séance de cours où ils devaient contrôler le même patch à plusieurs.

Ces commentaires sont riches d'enseignements pour nous, dans la mesure où ils témoignent *a posteriori* de l'utilisation du logiciel par les étudiants, de certaines limites du système, mais aussi des solutions *bricolées* qu'ils ont pu mettre en place pour les contourner. Ils témoignent aussi d'une certaine dimension ludique, relative à la communication entre les personnes et à la cocréation de contenus au sein des logiciels multi-utilisateurs, qui fait semble-t-il partie intégrante de la collaboration ; dimension ludique que nous avons d'ailleurs déjà eu l'occasion d'évoquer dans la première partie de ce document, lors des conclusions de l'étude de la collaboration au sein du forum avec la pratique du *Hippie Patching*[181].

7.2.6. Premiers retours

Les étudiants du cours nous ont fait remonter des améliorations réalisables à différents niveaux sur le logiciel. Du point de vue de l'objet *commentaire*, ils nous ont notamment dit qu'il serait souhaitable de savoir *qui* a créé le commentaire, *qui* et *quand* quelqu'un l'a édité, ou encore de disposer d'un système de notification automatique qui puisse transmettre par exemple un courrier électronique à la personne désignée au sein de l'objet. Plusieurs étudiants nous ont aussi fait la remarque que le langage de Kiwi avait été suffisant pour mettre en place la plupart de leurs idées musicales durant le cours mais qu'il manquait plusieurs objets indispensables pour qu'il puisse être utilisé confortablement. Enfin, l'interface du *Document Browser* demanderait aussi selon eux à être améliorée par l'ajout de sous-dossiers ou d'un mécanisme de recherche de document qui permette de mieux s'y retrouver. Nous essayerons donc de

[181] Voir 2.1.1.1 (p. 37).

prendre en considération ces différents retours dans le cadre du développement des prochaines versions de l'application.

7.3. Représentations musicales

L'atelier de composition du département de Musique de l'Université Paris 8[182] a toujours été depuis sa création un terrain d'expérimentation, notamment pour les différents projets développés au sein du CICM. Cela l'a été à partir de 2012 pour la bibliothèque HOA, avec laquelle la spatialisation de nombreuses pièces d'étudiants a été élaborée, et ça l'est désormais aussi pour Kiwi. Dans ce cadre, nous souhaitons revenir brièvement sur la création de la pièce mixte pour guitare et électronique composée par Bastien Loizillon, parce qu'elle représente la première pièce mixte publique réalisée et exécutée avec le logiciel Kiwi, mais aussi parce qu'elle ouvre à des perspectives intéressantes dans le cadre de la mise en place d'un contrôle collaboratif du patch.

7.3.1. Pièce mixte avec contrôle collaboratif du patch

Deux représentations de cette pièce ont été données, la première à l'Université Paris 8 le 14 juin 2018 dans le cadre de l'atelier de composition, et la seconde lors du concert de l'Ensemble MG2 du 15 juin 2018 à l'institut *Cervantes* à Paris[183]. La pièce de Bastien est une pièce mixte pour guitare et électronique en temps réel. Elle comporte plusieurs modules de traitement et de synthèse sonore indépendants spatialisés[184].

---

[182] Associé à la Maison des Sciences de l'Homme Paris-Nord et dirigé par José-Manuel Lopez-Lopez et Anne Sèdes.

[183] https://paris.cervantes.es/fr/ [consulté le 20/09/18].

[184] La pièce a été revue et corrigée entre les deux représentations par le compositeur qui a notamment ajouté une spatialisation sur huit canaux au lieu de deux et de nouveaux traitements. Le choix d'une diffusion spatiale octophonique séparant chaque module de traitement sur un canal a aussi été fait pour permettre à chaque intervenant de mieux s'entendre.

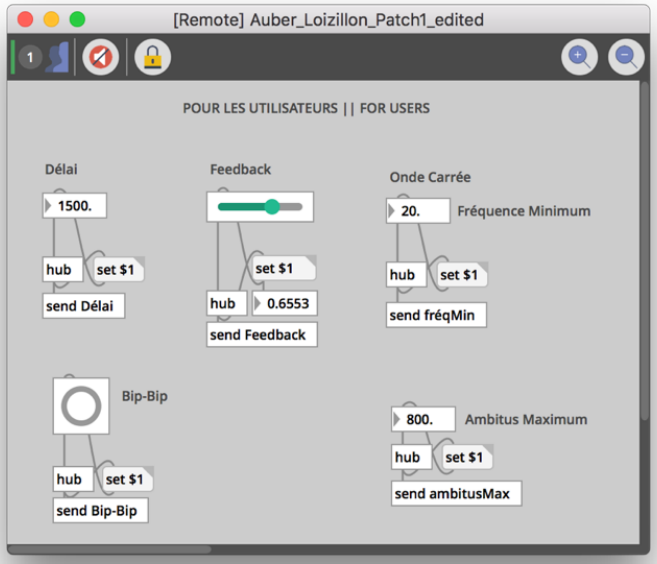

Figure 68 – Capture d'écran exposant les différents paramètres de contrôle du patch accessibles au public durant l'exécution de la pièce mixte de Bastien Loizillon[185].

Dans le cadre des deux représentations, le public était invité à interagir avec le patch et à faire varier les paramètres de jeu du patch proposés par le compositeur en temps-réel, et de manière collective durant l'exécution de la pièce [Figure 68]. Pour cela, des personnes pouvaient se connecter au même patch qui était en train d'être exécuté sur scène, depuis une autre machine, dans la même pièce ou même depuis l'extérieur, et agir ainsi directement à distance. En ce sens, nous nous situons ici dans la première colonne de la matrice d'Ellis exposée dans la première partie de ce document [Tableau 1] : « même moment / même lieu » et « même moment / lieux différents ». La communication avec le patch central s'effectuait à travers l'objet *hub* qui permet, grâce notamment au patron de conception exposé à la [Figure 61, p. 217], de modifier des paramètres du patch et d'être notifié de leur changements en temps réel [Figure 68].

[185] La version du patch présenté ici correspond à une version éditée du patch du compositeur, que nous avons légèrement modifié pour rester en cohérence avec le patron de conception lié à l'objet *hub*, que nous avons exposé à la [Figure 61, p. 217].

Six personnes se sont prêtées au jeu dans le cadre de la représentation à l'institut *Cervantes*, dont quatre personnes dans la salle de concert et deux à l'extérieur – ce qui fait un total de huit personnes intervenant sur l'exécution de la pièce en temps réel si on ajoute le compositeur et l'interprète se situant sur scène. Selon les retours obtenus, l'expérience a été plutôt concluante sur le plan musical, le public semblant avoir apprécié cette nouvelle forme d'interaction musicale permise par le logiciel Kiwi. Néanmoins, ce concert a permis aussi de s'apercevoir de certains problèmes inhérents à ce type d'approche ou de pratique.

7.3.2. Limites de l'approche

Ces difficultés sont celles de la représentation du processus à l'œuvre auprès du public, de la représentation des actions de jeu au sein du patch et enfin de la latence induite par le système.

7.3.2.1. Représentation des actions de jeu

Les personnes présentes dans la salle étaient d'abord perplexes, et ne comprenaient pas *qui* étai(en)t à l'origine des différents sons qu'ils entendaient, aussi se sont-t-elles tournées vers l'écran principal où se trouvait le patch, mais aussi sur les autres ordinateurs portables connectés au même patch dans la pièce, pour tenter de saisir ce qui se passait. Nous sommes donc ici en présence d'un autre exemple qui pointe le besoin de recourir à une représentation graphique du patch dans ce type d'approche, notamment à destination du public, pour expliciter les processus mis en œuvre et de manière plus générale la démarche musicale[186].

[186] Notons que dans le cadre de la représentation musicale qui a eu lieu lors du festival *La DéMO* 2017, et que nous exposons un peu plus haut dans ce mémoire, un système de projection du patch sur scène avait été mis en place pour ces raisons.

Ce concert nous a permis de nous rendre compte aussi de la nécessité de mieux représenter les actions de jeu des utilisateurs au sein du patch. Il faut souligner le fait que les utilisateurs extérieurs à la salle de concert pouvaient voir les paramètres du patch évoluer au sein du patch au même titre que tous les participants connectés (grâce à l'emploi de l'objet *hub*) mais qu'ils étaient sourds aux changements qu'ils opéraient sur le patch dans la mesure où le traitement du son s'effectuait sur scène et non chez eux, et où aucune transmission à distance du signal audio n'avait été prévue pour l'occasion. D'autre part, un des principaux retours obtenus de la part des personnes connectées au patch depuis l'intérieur de la salle de ce concert a été de dire qu'ils avaient du mal à interagir avec le patch du fait qu'ils ne voyaient pas suffisamment clairement ce que faisaient les autres en même temps qu'eux sur le patch. Cela nous donne donc une piste de développement importante à considérer pour les futures versions de l'application. Le but serait d'offrir une visualisation plus explicite des actions des autres utilisateurs sur le patch sur le plan de son exécution et non seulement de sa conception. Faire en sorte que chacun puisse prédire au mieux ce que va faire l'autre, son intention, à travers la notification des actions de jeu des autres participants. En d'autres terme, offrir de nouveaux mécanismes de conscience de groupe liée à la tâche spécifique du contrôle du patch.

7.3.2.2. Latence du système

Le dernier problème que nous analysons est celui de la latence induite par le système. Bien que Kiwi permette la mise en place d'un jeu collaboratif distribué en temps réel, notamment via l'objet *hub*, le logiciel n'a pas été conçu pour cet usage à la base. Aussi, de nombreux problèmes peuvent se poser, comme celui de la réactivité du système face à son contrôle par plusieurs utilisateurs. Si le temps de réponse de l'application est plutôt court du fait de l'architecture répliquée, le temps de notification y est quant à lui bien plus long du fait que les modifications doivent d'abord passer par le serveur central avant d'être distribuée sur les différents sites, chez

les clients connectés[187]. Si le changement de valeur des paramètres se fait quasiment instantanément sur la machine où est exécutée le patch et gère le signal audio envoyé aux enceintes, il est beaucoup plus long pour les utilisateurs connectés parallèlement, qui doivent subir la latence d'un aller-retour au serveur avant que leur modification ne soit prise en compte sur le patch central et donc qu'elle ait une incidence sur le son produit au cours de la pièce. Si un délai d'affichage peut encore être acceptable dans des situations comparables, il l'est beaucoup moins quand il s'agit d'agir sur un son en temps réel. Des solutions doivent donc pouvoir être trouvées pour améliorer les performances de l'application et réduire ces temps de latence en proposant notamment d'autres types de communications, en pair-à-pair par exemple, pour s'adapter à ce genre de situations où plusieurs personnes contrôlent un même patch en temps réel *au même endroit* et *en même temps*. Certains défendent néanmoins l'idée que la latence induite par le réseau n'a pas forcément à être subie et peut au contraire prendre pleinement part à la dimension créatrice. C'est, par exemple, le cas du compositeur Atau Tanaka : « Je trouve que le temps de délai sur Internet est plutôt intéressant et je le pense comme une sorte d'acoustique unique liée à ce média… Plutôt que de jouer de la musique existante sur cette nouvelle base de temps, ce qui me semble intéressant est de trouver un langage musical qui puisse fonctionner sur cet axe temporel… »[188]. Ce serait donc peut-être aussi aux compositeurs de trouver de nouvelles manières de composer en s'appropriant les modalités de jeu proposées par l'outil.

---

[187] Le temps de réponse est le temps qu'il faut entre le moment où un utilisateur effectue une action et la voit apparaître ou perçoive son effet sur sa propre interface. Le temps de notification est le temps que met une action à être propagée aux autres utilisateurs sur le réseau. Ces deux notions ont été détaillées à la section 3.1 (p. 88) de ce document.

[188] Citation de l'artiste Atau Tanaka dans une interview avec Golo Föllmer, reprise dans [Barbosa, 2003] et traduite de l'anglais : « I find Internet time delay rather interesting and I think of it as a kind of unique acoustic of this media... [R]ather than to play existing music on this new time basis, what is interesting to me is trying to find a musical language that works on this time axis… »

7.4. Présentations et ateliers

Deux présentations, accompagnées de démonstrations ont été faites dans le cadre des Journées d'Informatique Musicale[189]. La première, qui s'est déroulée lors de l'édition 2017 à Paris, a permis à l'équipe de montrer l'état d'avancement de l'application à ses débuts et son fonctionnement général à la communauté musicale présente [Paris & al., 2017]. La seconde, qui s'est déroulée à Amiens en 2018, était dédiée à l'exposition des premiers résultats de l'expérience pédagogique menée dans le cadre de la classe pilote réalisée avec Kiwi comme support à l'Université Paris 8 dont nous avons aussi parlé dans ce cadre [Galleron & al., 2018].

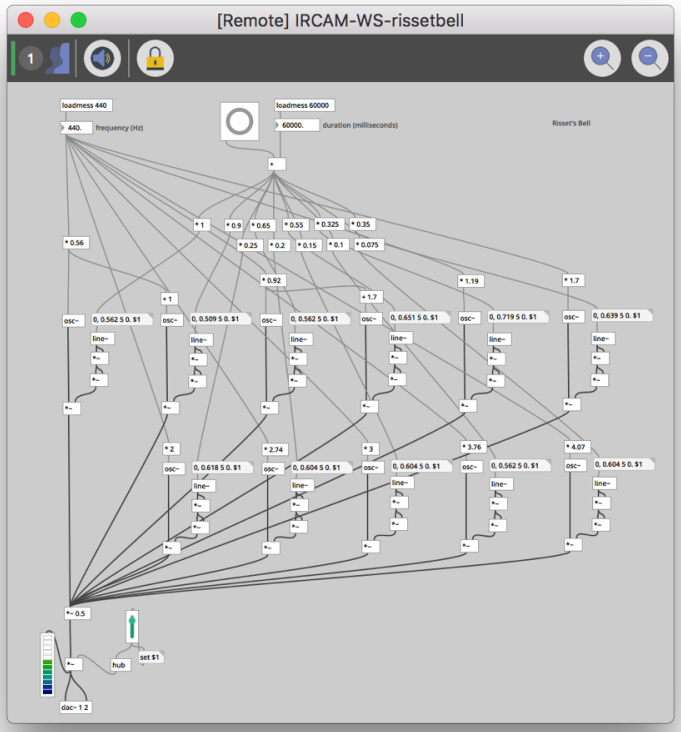

Figure 69 – Capture d'écran d'un patch Kiwi de synthèse fondé sur le modèle des cloches de Risset, réalisé par A. Bonardi dans le cadre des ateliers du Forum IRCAM.

[189] Les articles relatifs à ces deux présentations ont tous deux été récompensés par l'attribution du prix *jeune chercheur* de l'AFIM (Association Française d'Informatique Musicale).

Deux événements ont aussi été organisés à l'IRCAM dans le cadre du projet MUSICOLL. Le premier s'est fait sous la forme d'une présentation générale de ses enjeux et s'est accompagnée d'une démonstration générale du logiciel lors de la série de séminaire *Ircam Talk* le 8 février 2018[190].

Une seconde présentation, sous la forme d'un atelier participatif a ensuite été réalisée lors des *Ateliers du Forum IRCAM* qui se sont déroulés du 7 au 9 mars 2018[191]. Plusieurs patchs de traitement et de synthèse sonore ont été créés dans ce cadre. Parmi les patchs développés, il y a eu un patch de synthèse fondé sur le modèle des cloches de Risset [Figure 69], un *pitch-shifter* fondé sur une technique de *time-stretching*, de la synthèse par modulation de fréquence, un *flanger* ou encore un traitement de granulation[192]. Nous avons donc par-là pu valider le potentiel créatif et musical offert par l'application Kiwi.

Enfin, l'exposition des développements les plus récents au sein de l'application a été faite lors d'une présentation générale de Kiwi donnée par A. Bonardi et P. Guillot à l'occasion d'une table ronde ouverte réalisée durant la première conférence internationale dédiée au langage Faust qui a eu lieu les 17 et 18 juillet 2018 à l'Université Gutenberg à Mainz en Allemagne[193].

---

190 http://forumnet.ircam.fr/fr/event/seminaire-recherche-et-creation-kiwi-vers-un-environnement-de-creation-musicale-temps-reel-collaboratif/ [consulté le 16/09/18].

191 https://www.ircam.fr/agenda/les-ateliers-du-forum-1/detail/ [consulté le 16/09/18].

192 Ces patchs ont été créés sur le serveur en ligne, en grande partie par Alain Bonardi, et sont aujourd'hui accessibles et consultables à partir de la fenêtre *Document Browser* du logiciel Kiwi [vérifié le 12/09/18].

193 http://www.ifc18.uni-mainz.de/ [consulté le 16/09/18].

7.5. Intégration future de Kiwi dans un MOOC

Kiwi sera aussi intégré, à partir de novembre 2018, comme support pédagogique au MOOC intitulé « La musique au-delà du numérique »[194], co-construit par le CICM/Université Paris 8 (responsable pédagogique : A. Sèdes) et l'INA-GRM[195] (responsable pédagogique : D. Saint-Martin).

Cet enseignement en ligne sera adapté à un public diversifié de musiciens, d'artistes, d'étudiants en composition, en musique et en arts, d'enseignants ou de musicologues. Il se déroulera sur six séances de cours en ligne, et aura pour objectif :

- L'approfondissement des connaissances relatives à la musique électroacoustique et à l'évolution des techniques liées à la création musicale audionumérique.
- Le développement des capacités d'écoute et de créativité sonore et musicale.
- L'acquisition d'un niveau technique correspondant aux usages des métiers du son et de la création musicale.

La quatrième séance de ce cours sera dédiée à la synthèse sonore et à la composition du son en temps réel. Dans ce cadre, l'équipe pédagogique a prévu de permettre aux apprenants d'expérimenter quelques techniques actuelles de synthèse modulaire sur ordinateur avec le logiciel Kiwi. Les participants seront donc, dans un premier temps, invités à prendre en main l'application en téléchargeant la documentation, les tutoriels et les patchs d'aide pour comprendre le fonctionnement des différents objets de l'environnement logiciel et pouvoir appréhender un principe de programmation visuelle qui sera pour beaucoup encore inconnu. En

[194] https://www.fun-mooc.fr/courses/course-v1:parislumiere+142001+session01/about [consulté le 18/09/18].

[195] https://inagrm.com/fr [consulté le 18/09/18].

s'inspirant d'extraits de pièces de Stockhausen, Risset et Chowning, les apprenants seront invités à programmer un patch permettant d'enregistrer un moment musical d'une durée de vingt secondes en utilisant des oscillateurs, leur variable fréquentielle, un amplificateur, une sortie audio et un enregistreur de fichier son[196]. Pour les aider dans cette tâche, une collection de patchs réalisés spécialement pour le MOOC ont été créés [197]. Les participants y trouveront un exemple de générateur de sources sonores et d'enregistreur, un exemple de lignes à retard, un exemple de granulateur, un exemple inspiré du fragment *Strukture X* de *Kontakte* de K. Stockhausen. Un exemple inspiré de la synthèse des cloches de J-C. Risset est aussi fourni ainsi qu'un autre inspiré de la modulation de fréquence de J. Chowning.

L'équipe pédagogique prévoit un concert public de présentation à l'issue de ce cours à partir d'une sélection de pièces produites par les participants. Les retours liés à cette expérience seront pour nous précieux afin d'évaluer la solution Kiwi à plus grande échelle.

8. Perspectives

Nous venons d'évoquer dans le chapitre précédent une série de cas d'utilisation du logiciel Kiwi, dans un contexte à la fois pédagogique et de création musicale qui se sont déroulés au cours et dans le cadre le projet MUSICOLL mais qui s'étendent aussi au-delà. Ces différents projets ont montré l'intérêt des utilisateurs pour le logiciel, son potentiel musical et son adaptation au contexte de l'enseignement de la musique temps réel. Ils ont aussi fait remonter un certain nombre de limites à l'application. Dans ce chapitre nous nous intéresserons donc aux

[196] Les objets *sf.record~* et *sf.play~*, permettant respectivement d'enregistrer et de jouer des fichiers sons depuis l'environnement Kiwi, ont été ajoutés à l'application à la version *v1.0.2* afin de soutenir cet exercice.

[197] Ces patchs ont été réalisés par l'équipe MUSICOLL et plus spécifiquement par P. Guillot durant son contrat post-doctoral au sein du projet.

perspectives du projet, telles que les retours obtenus de la part des utilisateurs et l'état actuel de l'application nous les laissent envisager. Certaines nouvelles fonctionnalités sont déjà à l'étude et assez avancées pour être bientôt intégrées à l'application, comme on le verra dans la prochaine section dédiée à l'intégration de Faust à l'environnement Kiwi (8.1). D'autres en revanche représentent des pistes de recherches plus ouvertes à court, moyen ou plus long terme que nous devrons encore mieux spécifier avant d'envisager un développement ou une intégration éventuelle aux prochaines versions de l'application. Dans la seconde et la troisième section de ce chapitre, nous exposerons des solutions visant à l'amélioration de la coordination de l'activité de conception de patch, pour qu'elle puisse s'exercer aussi bien de manière synchrone qu'asynchrone. Nous proposerons alors de nouveaux mécanismes de conscience de groupe (8.2) et des améliorations à réaliser dans la manière de gérer les sessions au sein du logiciel (8.3). Dans la dernière section, nous nous intéresserons plus spécifiquement aux perspectives liées à l'activité de jeu collaboratif au sein de Kiwi et aux différentes problématiques qui y sont relatives (8.4).

8.1. Intégration du langage Faust

L'une des plus grandes faiblesses de l'application Kiwi actuellement, d'un point de vue fonctionnel, est selon nous son manque d'extensibilité. Les logiciels de patching tels que Max ou Pure Data sont extensibles de plusieurs manières. Ils supportent les concepts d'encapsulation et d'abstraction, qui permettent la réutilisation de modules développés avec le langage graphique du logiciel ou encore une API et un SDK[198] qui autorisent le développement d'objets externes, c'est-à-dire non compris dans la distribution originale du logiciel. Kiwi ne supporte pour l'instant pas le concept d'abstraction ni d'encapsulation, et ne comporte pas d'API qui

[198] *Software Development Kit* (Kit de développement logiciel), https://github.com/Cycling74/max-sdk [consulté le 14/09/18].

permettrait d'ajouter dynamiquement de nouveaux objets au logiciel, et donc de nouvelles fonctionnalités ou traitements sonores au langage. L'utilisateur de Kiwi est donc en ce sens pour l'instant limité aux fonctionnalités statiques offertes par l'outil. L'une des perspectives futures de développement se situe donc ici. Néanmoins, pour pallier ce manque dans un premier temps et fournir une palette plus riche en terme de possibilités de création au sein de Kiwi, nous avons récemment étudié au sein de l'équipe la possibilité d'y intégrer le langage de programmation Faust[199]. Nous détaillerons dans un premier temps les enjeux relatifs à cette intégration, qui a pris la forme d'un nouvel objet, intitulé *faust~*, permettant d'éditer du code Faust au sein d'un patch. Ce développement a été pensé et spécifié en équipe, mais sa réalisation est principalement le fruit du travail de P. Guillot, engagé en qualité de post-doctorant au sein du projet MUSICOLL sur la période d'avril à août 2018. Nous détaillerons la mise en œuvre de cet objet dans un second temps[200].

8.1.1. Enjeux et pistes de recherche

Le principal enjeu de ce développement est, comme nous l'avons dit plus haut de permettre d'étendre de manière dynamique les possibilités offertes par le logiciel Kiwi. D'un point de vue pédagogique, l'intégration de Faust au sein de Kiwi permet aussi d'offrir de nouvelles pistes d'enseignement et servir de passerelle pour une initiation à ce langage ainsi qu'à la programmation textuelle en général. Outre la complexité relative à ce nouveau langage pour des étudiants en musique assistée par ordinateur, qui ne sont souvent pas habitués à la programmation textuelle, ces derniers ont souvent aussi du mal à se lancer dans son

[199] http://faust.grame.fr/ [consulté le 14/09/18].

[200] Notons que si le développement de cette nouvelle solution est bien avancé, cet objet n'est pas encore supporté officiellement dans la version actuelle de Kiwi au moment de l'écriture de ce mémoire de thèse qui correspond à la *v1.0.2*. Il devrait néanmoins être intégré à la prochaine version.

apprentissage à cause de la relative complexité d'installation requise avant de pouvoir l'utiliser[201]. Le fait d'embarquer Faust dans Kiwi permet donc en ce sens de faciliter et d'accélérer sa prise en main en offrant une solution clef en main directement intégrée à un environnement logiciel de création musicale en temps réel.

Faust (*Functional Audio Stream*) est un langage de programmation fonctionnel dédié spécifiquement au traitement temps réel du signal et à la synthèse, ainsi qu'une bibliothèque de codes et d'outils développés par le GRAME[202] [Orlarey & al., 2009]. Les principaux attraits de l'écosystème Faust qui ont motivé son intégration au sein de Kiwi étaient le fait qu'il soit *open-source* et portable ; qu'il dispose d'une large communauté d'utilisateurs[203] qui contribue activement à la maintenance du noyau logiciel en le faisant évoluer ; qu'il comprend de nombreux tutoriels, exemples et modules qui rendent son appropriation plus aisée ; enfin par le fait qu'il assure une certaine pérennité aux traitements sonores réalisés dans la mesure où il permet aussi de générer une documentation mathématique de ceux-ci.

Plus qu'un simple langage, la bibliothèque Faust propose aussi une série d'outils logiciels qui permettent de générer des exécutables dans différents formats. Le compilateur Faust permet de traduire des instructions DSP, écrites en langage Faust, de manière optimisée dans divers langages de programmation (*e.g.* le C, le C++, le Java, le JavaScript, WebAssembly). Le code

---

[201] L'utilisateur peut par exemple être un peu perdu face aux nombreuses versions de Faust, à l'installation du compilateur ou encore de l'environnement FaustLive suivant la plateforme sur laquelle il se trouve. Le lien suivant décrit la procédure d'installation : https://ccrma.stanford.edu/~rmichon/faustTutorials/#setting-up-your-development-environment [consulté le 16/09/18].

[202] http://www.grame.fr/ [consulté le 14/09/18].

[203] Faust est devenu aujourd'hui très populaire dans le domaine du développement audio et du traitement du signal, comme peut en témoigner la première conférence internationale dédiée au langage Faust qui s'est tenue les 17 et 18 juillet 2018 à l'Université Gutenberg à Mainz en Allemagne. http://www.ifc18.uni-mainz.de/

généré peut ensuite soit être utilisé directement, sur un navigateur par exemple dans le cas du JavaScript ou du WebAssembly, soit être compilé en langage machine pour produire une application native autonome, une bibliothèque, un plugiciel sur diverses plateformes et suivant différents standards. L'écosystème Faust et les outils fournis permettent alors de générer facilement, à partir du même code Faust, des applications autonomes de bureau, des applications mobiles iOS ou Android, des plugiciels au format VST[204] ou AudioUnit[205] ou encore des modules de traitement du signal destinés à des logiciels hôtes tels que *SuperCollider*[206], Pure Data [Graef, 2007], ou encore Max [Orlarey & al., 2009].

Les perspectives de développement et d'intégration qui s'offraient à nous étaient alors les suivantes :

- Pouvoir utiliser un exécutable généré avec Faust au sein de Kiwi.
- *Transcrire* un patch Kiwi en langage Faust par sérialisation de chaque module de traitement du signal, pour pouvoir l'exécuter ensuite de manière autonome au sein de l'application de manière plus optimisée, l'exporter sous forme d'exécutable autonome, ou encore de code pour y être intégrer au sein d'une application tierce.
- Pouvoir utiliser du code Faust au sein de Kiwi pour générer un rendu audio, le partager avec d'autres personnes et l'éditer de manière collaborative.

La première solution consistait à ajouter une fonctionnalité à la bibliothèque Faust qui définit une nouvelle cible de déploiement du code Faust à destination de Kiwi, comme a pu le faire A. Graef à destination du logiciel Pure Data [Graef, 2007]. Celle-ci a vite été écartée dans la

---

[204] https://www.steinberg.fr/fr/produits/vst.html [consulté le 07/09/18].

[205] https://developer.apple.com/documentation/audiounit [consulté le 07/09/18].

[206] https://supercollider.github.io/ [consulté le 07/09/18].

mesure où l'application Kiwi est toujours au stade expérimental et ne dispose pas encore d'API. Il serait donc trop précipité à l'heure actuelle d'effectuer un développement qui irait dans ce sens au risque de ne pas être pérenne. La seconde représente un réel intérêt dans la mesure où elle permettrait de réaliser un pont depuis le langage Patcher de Kiwi vers les formats plus standard de l'industrie audio en permettant la *transcription* d'un patch en langage Faust, puis en langage textuels intégrables à des applications ou plugiciels tiers. Cette idée n'est pas nouvelle. Elle nous vient en fait de la technologie *Gen* introduite au sein de l'environnement Max à partir de la version 6[207]. Cette technologie vise à créer des traitements, notamment sonores, en les programmant visuellement et à les transcrire via le langage *GenExpr*[208] en langages de programmation textuels plus traditionnels dans le même but. Cependant, le langage *GenExpr*, tout comme le logiciel Max, reste propriétaire. Aussi, une nouvelle approche de ce type, mais qui soit libre, gratuite et fondée sur le langage Faust, pourrait alors se révéler bénéfique à l'ensemble de la communauté de l'informatique musicale. Néanmoins, sa mise en œuvre est complexe, elle demanderait des spécifications détaillées et un temps de développement relativement conséquent. Aussi avons-nous opté pour la troisième solution évoquée plus haut qui se révélait, bien que complexe, plus réalisable dans le cadre temporel qui était dédié au développement vers la fin du projet.

Cette mise en œuvre s'inspire de l'objet externe *faustgen~* créé par l'équipe du GRAME pour l'environnement Max[209]. L'objet *faustgen~* permet de charger dynamiquement du code Faust, de le compiler à la volée en s'appuyant sur les bibliothèques *libfaust* et LLVM[210], puis de

---

[207] https://docs.cycling74.com/max7/maxobject/gen~ [consulté le 20/09/18].

[208] https://docs.cycling74.com/max7/vignettes/gen_genexpr [consulté le 20/09/18]

[209] http://www.grame.fr/logiciels/faustgen [consulté le 20/09/18].

[210] Low Level Virtual Machine

l'exécuter en temps réel dans Max [Letz & al., 2013]. Il autorise le chargement de fichiers en langage *faust* et comporte aussi un éditeur de texte qui permet de modifier le code exécuté dynamiquement. L'idée a donc été de créer un nouvel objet, baptisé *faust~*, qui puisse supporter l'édition et l'exécution de code Faust au sein de l'environnement Kiwi. L'enjeu était, d'une part de pouvoir utiliser les modules de traitement du signal déjà présents au sein des bibliothèques Faust, et d'autre part de pouvoir, dans le contexte collaboratif de Kiwi, éditer du code Faust à plusieurs et ouvrir ainsi à de nouvelles perspectives d'utilisation du langage, notamment sur un plan pédagogique.

8.1.2. Présentation et mise en œuvre de l'objet *faust~*

Ce développement a dû faire face à deux difficultés principales : d'une part l'intégration du moteur de rendu sonore et de compilation de code Faust au sein de l'environnement ; et d'autre part de permettre une édition collaborative du code entre plusieurs participants à une session. La première difficulté, plutôt d'ordre technique, a été de supporter les mêmes fonctionnalités sur les différentes plateformes ciblées, puis d'intégrer les dépendances nécessaires à ce développement au sein du projet Kiwi. Pour cela, P. Guillot a procédé en deux étapes. Un premier projet, intitulé *pd-faustgen*, a consisté à faire un portage de l'objet *faustgen~* initialement développé pour Max, vers l'environnement Pure Data[211]. La création de cet objet a permis de rencontrer et régler les principaux problèmes liés à l'embarcation de Faust dans une application tierce, en l'occurrence ici un objet externe, sur l'ensemble des plateformes supportées par Kiwi à savoir Mac, Windows et Linux. Une fois ce développement réalisé et la logique comprise, le code pu ensuite être adapté et intégré à Kiwi.

[211] Le code relatif à ce développement est disponible à l'adresse suivante : https://github.com/CICM/pd-faustgen [consulté le 20/09/18].

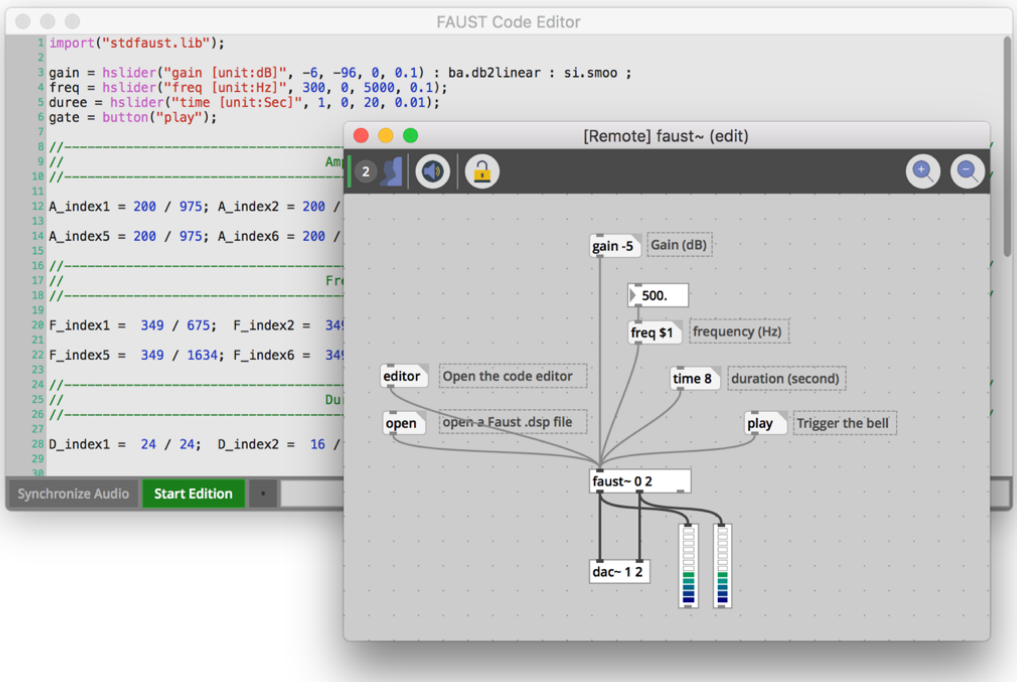

Figure 70 – Capture d'écran montrant une utilisation de l'objet *faust~* au sein de Kiwi, les différents objets contenus dans le patch servent à contrôler l'objet ou les paramètres du traitement.

L'objet *faust~* de Kiwi, tel qu'il est actuellement mis en œuvre, est une boîte de type non-graphique qui requiert à sa création deux arguments définissant le nombre d'entrées et le nombre de sorties de type signal. L'objet adapte alors son nombre d'entrées et de sorties en fonction. L'objet comporte au minimum une entrée qui permet de communiquer avec l'objet en lui envoyant des messages, notamment pour changer dynamiquement les paramètres actifs du rendu sonore. Il comporte aussi une sortie supplémentaire de type message qui permet de récupérer la valeur des paramètres passifs du moteur audio au sein du patch[212]. La figure ci-dessus expose en premier plan un patch Kiwi dans lequel se trouve l'objet *faust~* et différentes boîtes-message destinées au contrôle du traitement sonore exécuté [Figure 70]. Lorsque l'utilisateur envoie le message *open* à l'objet, une fenêtre de dialogue système s'ouvre lui

[212] Les paramètres passifs sont des paramètres qui génèrent des valeurs, comme la valeur courante du signal.

permettant de charger des fichiers au format Faust depuis la machine locale. Lorsqu'un nouveau fichier est chargé au sein de l'objet, le code Faust est compilé par l'objet pour obtenir un rendu sonore au sein du patch. Les informations relatives au code (texte, mais aussi paramètres et options de compilation du code) sont stockées au sein du document Kiwi représentant le patch. Cela a deux conséquences. La première est que de cette façon, le texte n'a pas à être chargé de nouveau au sein de l'objet à la réouverture du patch. La seconde est que ces informations sont alors synchronisées chez chaque participant lorsqu'ils sont connectés à une même session depuis le serveur. Un utilisateur peut alors charger un fichier depuis sa machine pour le rendre disponible à un autre au sein d'un patch partagé. En arrière-plan de la [Figure 70], on peut apercevoir l'éditeur de code Faust intégré qui s'ouvre lorsque l'objet reçoit le message *editor*.

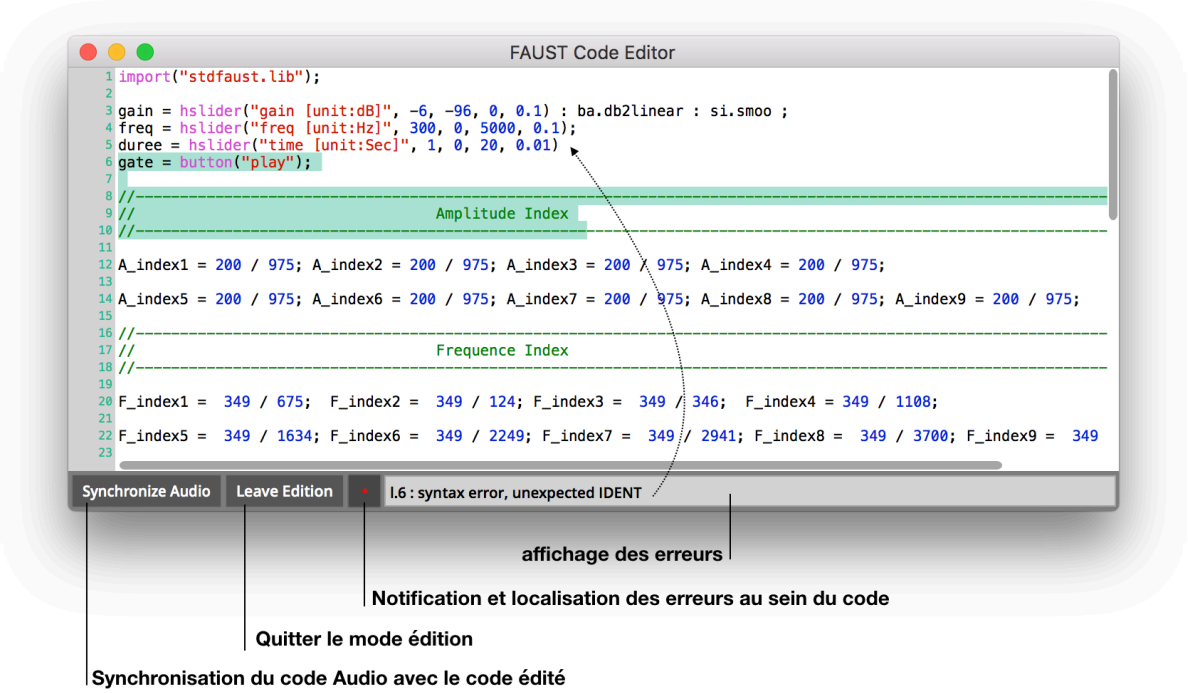

Figure 71 – Capture d'écran de l'interface graphique d'édition de code Faust embarquée au sein de l'objet *faust~* de Kiwi. La partie de code exposée dans cette capture provient d'une mise en œuvre en langage Faust du modèle de cloches de Risset, réalisée par le compositeur et doctorant João Svidzinski.

Cet éditeur permet de modifier le rendu sonore de l'objet *faust~* par l'édition du code Faust à partir de l'interface de Kiwi. Le code édité est alors mis à jour de la même manière au sein du

modèle de données, ouvrant alors à une édition collaborative de son contenu à travers cette interface. L'éditeur de code de l'objet *faust~* dans Kiwi, par rapport à celui présent au sein de l'objet *faustgen~* de Max, comporte plusieurs améliorations fonctionnelles. Il gère notamment la coloration syntaxique du code spécifique au langage Faust[213], et permet l'affichage des erreurs directement sur l'interface [Figure 71]. À chaque fois que le texte est modifié au sein de l'éditeur, l'objet recompile le code Faust *à la volée*, de façon à pouvoir présenter les erreurs d'édition éventuelles à l'utilisateur. Lorsque l'utilisateur qui édite le texte clique sur le bouton « Synchronize Audio », le code courant est compilé par l'objet puis transmis au moteur audio qui actualise alors le rendu sonore de l'objet. Lors d'une session connectée, tous les participants peuvent mettre à jour le code d'un objet *faust~* et synchroniser ainsi le rendu sonore de l'objet sur toutes les machines connectées. Si un utilisateur ouvre l'éditeur alors qu'un autre est en train d'éditer son contenu il voit alors en temps réel les modifications apportées aux texte dans la mesure où le code est transmis par l'intermédiaire du modèle à tous les participants en temps réel. Mais que se passe-t-il lorsque plusieurs personnes tentent d'éditer en même temps ce code ? Autrement-dit comment gérer les problèmes de concurrence liés à l'édition collaborative de texte au sein de cette interface ?

Cette problématique a représenté en fait la seconde difficulté de cette mise en œuvre dans la mesure où nous avions à gérer un cas encore non-résolu au sein de Kiwi. Au cours de la présentation de l'application dans la seconde partie de ce mémoire, nous avons pu noter que le texte au sein des objets, que ce soit au niveau des boîtes ou des objets commentaires n'était mis à jour chez les autres participants qu'à partir du moment où l'utilisateur mettait fin à son action

[213] La mise en place de la coloration syntaxique repose sur un code d'Oliver Larkin : https://github.com/olilarkin/juce_faustllvm [consulté le 16/09/18].

d'édition[214]. Aussi, deux utilisateurs peuvent éditer le même objet en même temps au sein de Kiwi, mais la gestion de la concurrence ne s'applique pas au contenu textuel de l'objet. Lorsqu'un utilisateur valide l'entrée d'un texte au sein d'un commentaire, ce texte est mis à jour chez tous les participants, écrasant alors le texte qu'est en train d'éditer un autre utilisateur sur le même objet. Si ce comportement peut être acceptable dans le contexte d'édition d'une boîte où le texte est généralement court, le code contenu ici au sein de l'éditeur est destiné à être beaucoup plus important. Aussi l'écrasement de celui-ci par une action d'édition venant d'un autre utilisateur est nettement moins supportable. Pour régler ce problème nous devions donc gérer la question de l'édition collaborative de texte en temps réel. La bibliothèque *flip* n'a pas été conçue initialement pour le gérer. Même si un développement allant dans ce sens avait été évoqué et envisagé dans le cadre du projet MUSICOLL, il n'a pas encore été effectué à l'heure actuelle et nous aurions donc dû soit intégrer une bibliothèque tierce, soit recréer tout un système, ce qui n'était alors pas envisageable au vu de la complexité de la tâche. D'autre part, le contrôle de la concurrence n'est pas le seul problème lié à l'édition collaborative de texte en temps réel. En effet, comme nous l'avons vu à plusieurs reprises au cours de ce mémoire, des adaptations doivent souvent se faire au niveau de l'interface pour supporter le fait d'être manipulées ou éditées par plusieurs personnes en même temps. La création d'un éditeur de texte multi-utilisateur et collaboratif en temps réel est une tâche complexe qui nécessite un temps de développement conséquent. L'éditeur doit supporter notamment plusieurs curseurs ou encore notifier de la sélection des autres participants au sein de l'interface, comme nous l'avons vu à la [Figure 39, p. 167]. Or la bibliothèque *Juce*, que nous utilisons notamment pour le développement de l'interface graphique, ne supporte pas ces fonctionnalités au sein de son

[214] Voir sous-section 4.3.2 (p. 142) de ce document.

composant d'édition textuelle. Comme nous ne disposions plus de ressources suffisantes à dédier à cette tâche vers la fin du projet, il nous a donc fallu trouver une approche alternative.

On peut qualifier la solution actuellement mise en place pour contrôler l'édition concurrente du document de pessimiste[215], dans la mesure où le premier utilisateur qui démarre l'édition du document et lui-seul, dispose des droits en écriture, et ce jusqu'à ce qu'il laisse la main. Cette approche nous a permis de mettre en œuvre rapidement un système qui fonctionne. Néanmoins, elle n'est généralement pas idéale dans la mesure où elle pose de nombreux problèmes pratiques. Une personne qui a la main sur le contenu peut empêcher les autres d'y accéder. Si la personne garde la main trop longtemps, s'absente ou encore oublie de libérer l'accès, les autres utilisateurs sont par exemple bloqués et ne peuvent pas modifier comme ils le souhaiteraient son contenu.

Pour contourner ces problèmes nous avons donc décidé de mettre en place une approche plus flexible permettant à la fois aux utilisateurs d'observer les modifications apportées à un document par un autre, mais aussi de pouvoir prendre la main s'ils le souhaitent sur l'édition. La figure ci-dessous expose une vue de l'éditeur de code tel qu'il apparaît chez un utilisateur lorsqu'un autre est actuellement en mode édition.

[215] Voir section 3.5 (p. 94) de ce document.

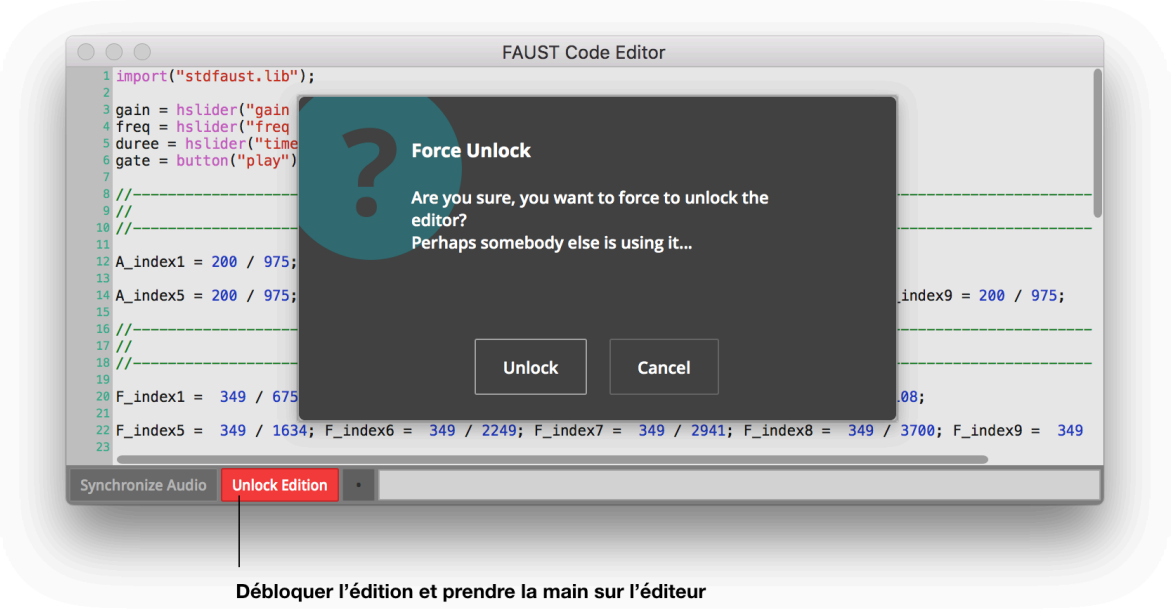

Figure 72 – Capture d'écran de l'éditeur de code de l'objet *faust~*, tel qu'il apparaît à un utilisateur lorsque quelqu'un d'autre est en train d'éditer le texte.

Ce mode ne lui permet pas d'éditer le texte, en revanche, il peut suivre son édition en temps réel dans la mesure où le texte tapé est transmis à tous les utilisateurs connectés à la session. La manière dont a été conçue l'interface permet aussi à l'utilisateur de prendre la main de manière forcée sur le code en appuyant sur le bouton « Unlock Edition » [Figure 72]. Une fenêtre contextuelle s'affiche alors pour l'informer que cette action va avoir pour conséquence de faire perdre la main à l'éditeur courant à son profit. Si l'utilisateur accepte, l'éditeur est déverrouillé de son côté, se retrouve dans la configuration présentée plus haut [Figure 71, p. 256], et laisse alors l'utilisateur la possibilité d'apporter les modifications qu'il souhaite au code.

Ce développement spécifique permet encore d'ouvrir de nouvelles pistes de recherche à explorer, notamment pour permettre d'offrir une collaboration synchrone du code au sein de

l'éditeur[216]. Néanmoins, l'état actuel de cet objet permet déjà d'offrir de nouvelles perspectives de jeu. Aussi, après avoir effectué quelques tests supplémentaires, il devrait donc être intégré en l'état à la prochaine version de l'application. Cette nouvelle fonctionnalité a d'ailleurs déjà pu être appréciée par la communauté Faust présente à la première conférence internationale dédiée au langage qui s'est tenue à l'Université Gutenberg de Mainz en Allemagne en Juillet 2018, où l'équipe a pu en faire une première démonstration. Cet objet sera également utilisé sur six séances dans le cadre du cours intitulé *langages de programmation en informatique musicale 2* au second semestre 2018-2019, pour renouveler l'enseignement de Faust. Nous espérons donc que l'objet pourra ainsi bénéficier aux étudiants et plus généralement à tous les utilisateurs de Kiwi en leur permettant de se former au langage et de l'exploiter à des fins musicales, dans la même optique d'apprentissage par le *faire-ensemble* qui est propre à l'application.

8.2. Conscience de groupe

Dans la seconde partie de ce document, lors de l'étude des aspects liés à la conception de l'espace de coordination de l'application Kiwi, nous avons exposé les éléments de l'interface partagée du patch qui ont permis d'apporter une première réponse à la problématique de la conscience de groupe entre les participants à une session. Les participants peuvent savoir qui est présent au sein du patch ou encore coordonner leurs actions en temps réel en observant les sélections des autres personnes connectées[217]. Les informations de ce type se révèlent cruciales pour travailler efficacement à plusieurs sur un même document, notamment lors de l'activité de conception du patch lorsque les participants interagissent en temps réel. Néanmoins, les

[216] Notons que cet enjeu se retrouve aussi dans le fait de permettre à plusieurs utilisateurs d'éditer par exemple de manière collaborative un même objet *commentaire*.

[217] Voir notamment les sous-sections 5.1.1 (p. 164) et 5.1.2 (p. 166) de ce document.

sélections ne suffisent pas toujours pour traduire l'activité ou la localisation des utilisateurs au sein du patch. En effet, si un utilisateur ne sélectionne aucun élément, les autres n'ont aucun indice quant à sa localisation. De même, il se peut qu'un utilisateur ait sélectionné un objet mais soit en fait en train de visualiser une autre zone du patch, là encore les autres utilisateurs auront un indice erroné sur sa véritable localisation. D'autre part, le fait qu'un objet soit sélectionné par un utilisateur ne renseigne pas les autres de l'intention exacte de ce dernier, c'est-à-dire sur l'action qu'il s'apprête à effectuer ou qu'il est en train d'effectuer. Aussi, quelles solutions pourrait-on envisager de mettre en place afin de mieux traduire ces actions à travers l'espace virtuel du patch ? Les deux sous-sections suivantes, (8.2.1) et (8.2.2), s'inspirent des travaux réalisés par le groupe de travail *GroupLab* de l'Université de Calgary au Canada qui a étudié en profondeur ces questions. Nous tenterons alors d'étudier et de transposer certaines des solutions concrètes proposées notamment par la bibliothèque *GroupKit* [Roseman & Greenberg, 1992, 1996], [Greenberg & Roseman, 1998], au contexte de Kiwi. Une autre dimension à la conscience de groupe est celle liée à la collaboration asynchrone. Le logiciel Kiwi permet à plusieurs personnes de collaborer en temps différé, c'est-à-dire de manière séquentielle, sur le même document. Néanmoins, dans ce contexte précis, les modalités de conscience de groupe que nous avons jusqu'ici mises en place ne s'appliquent plus dans la mesure où les participants ne sont plus connectés en même temps sur le logiciel et n'ont donc plus conscience des modifications apportées par chacun des collaborateurs. Cette problématique est celle de la conscience du changement, formalisée notamment par des auteurs tels que L. McCaffrey [McCaffrey, 1998]. L'enjeu se situe ici dans la possibilité d'offrir aux utilisateurs les moyens de prendre en considération les changements apportés par les autres à un document ou à une partie du document durant leur absence. Nous étudierons ces questions dans la troisième sous-section (8.2.3).

8.2.1. Partage du pointeur de souris

Des études menées sur des groupes de plusieurs personnes travaillant sur un espace partagé ont montré que les gestes constituaient environ un tiers de l'activité du groupe lorsqu'il s'agissait d'exprimer des idées, et presque la moitié quand il s'agissait d'accompagner une interaction [Tang, 1991]. Sur une interface physique (*e.g.* un tableau blanc de dessin) les utilisateurs peuvent pointer du doigt une partie pour attirer l'attention de l'audience sur celle-ci. Dans le cadre d'un cours, un professeur peut aussi le faire en pointant un élément au tableau ou sur un écran visible par toute la classe. Les gestes sur les interfaces graphiques des logiciels de bureau peuvent se faire de façon similaire par l'intermédiaire du pointeur de souris qui se substitue alors aux mains. Si plusieurs personnes regardent le même écran, ce pointeur peut servir à indiquer un objet ou un groupe d'objet particulier en le pointant ou en l'entourant. En revanche, quand les participants se trouvent dispersés physiquement, ils ne peuvent plus se raccrocher à cette information qui est alors manquante. Même si les utilisateurs gardent un contact oral et/ou visuel entre eux, par l'intermédiaire d'un logiciel de visioconférence par exemple, des phrases prononcées telles que : « *on devrait revoir cette partie !* », ou encore « *à quoi sert cet objet ?* » ne fonctionnent plus aussi bien et n'ont plus le même impact sans l'intermédiaire d'un geste associé à un référent commun. Comment donc remédier à ce problème et apporter à l'utilisateur une meilleur conscience et appréhension de l'espace partagé ?

Une solution envisageable pourrait être de s'inspirer de la notion de *télépointeur*, décrite en détail dans [Greenberg & al., 1996], [Roseman & Greenberg, 1996]. Un *télépointeur* représente le pointeur d'un autre participant à la session sur l'espace de travail virtuel d'un utilisateur. Si l'on adapte ce procédé à Kiwi, cela consisterait à transmettre la position de la souris en temps réel à tous les autres participants, qui verraient alors s'afficher autant de pointeurs que d'utilisateurs connectés au document. Cette fonctionnalité ne serait pas d'une grande

complexité de mise en œuvre au niveau du modèle, qui pourrait la supporter comme une donnée non persistante. Dans le modèle de données relatif au patch, tel qu'il est actuellement conçu, nous pouvons stocker de manière isolée des informations relatives et spécifiques à chaque utilisateur [Figure 21, p. 130, vignette 1, ligne 5]. Nous nous en servons actuellement pour stocker des informations sur les différentes vues du patch, telles que les sélections ou encore les valeurs de zoom. Il nous suffirait alors, pour supporter le concept de *télépointeur*, de rajouter deux variables membres au modèle de la vue, qui correspondraient aux coordonnées du pointeur de la souris de l'utilisateur[218]. L'application mettrait alors à jour automatiquement et à intervalle régulier ces valeurs. Notons que les coordonnées du pointeur doivent être relatives à l'origine du patch sur lequel l'utilisateur a le focus, et fonction du niveau de zoom, pour que l'endroit pointé par un utilisateur reste cohérent par rapport à la vue d'un patch chez les autres. Le support de cette fonctionnalité dans Kiwi supposerait aussi de repenser plusieurs choses au niveau de l'interface graphique. Cela impliquerait notamment de pouvoir distinguer graphiquement les utilisateurs les uns des autres ; décision que nous n'avions pas prise lors de la mise en œuvre des sélections des objets et des liens dans le patch. En effet, la granularité au sein des sélections ne se situe pas au niveau de l'utilisateur mais d'un groupe d'utilisateurs distants, la couleur des sélections affichée restant la même si un ou plusieurs utilisateurs différents sélectionnent des éléments.

[218] Rappelons qu'une vue au sein du modèle de données appartient à un utilisateur spécifique.

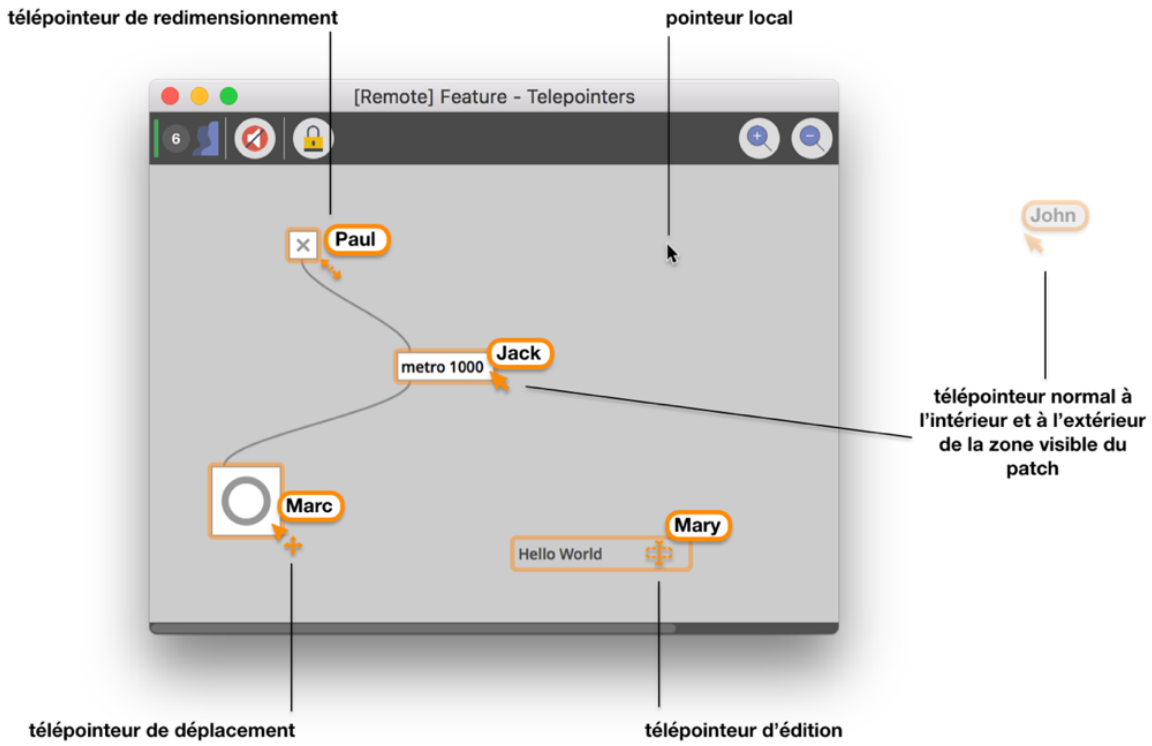

Figure 73 – Capture d'écran d'une maquette d'interface graphique proposant une mise en œuvre du concept de *télépointeur* sémantique au sein d'un patch Kiwi. Le nom de chaque participant est affiché au-dessus de leur pointeur virtuel respectif. L'icône du pointeur est surchargée en fonction du type d'action qu'est en train de réaliser chaque utilisateur.

Néanmoins, afficher un pointeur par utilisateur n'a de sens que si on est clairement capable de distinguer un utilisateur d'un autre, et en l'occurrence leur pointeurs respectifs, ce système ne pourra donc être viable que si nous usons de couleurs spécifiques, ou au minimum si nous affichons une information, telle qu'un nom, permettant d'identifier indépendamment chaque utilisateur. L'illustration ci-dessus propose une version maquettée de l'interface du patch telle qu'elle pourrait se présenter si nous supportions la fonctionnalité de *télépointeur* au sein de Kiwi [Figure 73]. Dans cette version, les pointeurs gardent une couleur similaire à celle des sélections, en revanche, le nom situé à côté permet bien d'identifier chaque utilisateur. Dans la majorité des applications de bureau, le pointeur de la souris est surchargé, c'est-à-dire qu'il change d'apparence en fonction du contexte ou pour refléter l'action courante que l'on est en

d'effectuer sur notre machine. Ce type de pointeur est alors dit *sémantique*, dans la mesure où il apporte un sens à l'action en cours réalisée. En suivant le même principe, on pourrait alors aussi imaginer surcharger les pointeurs des autres participants pour donner une information supplémentaire sur leur activité au sein du patch. Le pointeur pourrait par exemple représenter un curseur lorsqu'un utilisateur est en train d'éditer un objet, comme dans le cas de l'utilisateur surnommé *Mary* [Figure 73], ou encore afficher une icône spécifique lorsqu'il redimensionne ou déplace un objet, comme dans le cas des utilisateurs surnommés *Marc* et *Paul* [Figure 73]. Les utilisateurs au sein d'un patch peuvent aussi se trouver sur une zone différente de celle que l'on visualise. Cette information pourrait alors être traduite par un alpha diminué de la couleur du pointeur, comme dans le cas de l'utilisateur *John* [Figure 73]. La mise en place des *télépointeurs* sémantiques nécessiterait aussi d'ajouter un champ au modèle de la vue pour stocker l'action courante de l'utilisateur afin que les autres puissent en être notifiés et l'observer.

Le principe de *télépointeur* pourrait encore être élargi et enrichi de nouvelles fonctionnalités. L'une d'entre-elle pourrait être d'envoyer une sorte de signal aux autres utilisateur pour attirer l'attention des autres utilisateurs sur une zone spécifique du patch. Ce signal pourrait être déclenché par un clic sur le patch d'un utilisateur et se manifester par une animation au sein de chacun des patchs des autres participants, les invitant ainsi à déplacer leur zone de visualisation à l'endroit d'où le signal provient. Cette fonctionnalité aurait pu améliorer selon nous les interactions entre les différents acteurs de l'expérience pédagogique du cours pilote de Kiwi à l'Université où les étudiants auraient alors pu plus facilement interpeller leurs pairs ou le professeur sur une zone spécifique d'un patch grâce à elle.

Notons que le mode de jeu pourrait aussi bénéficier de cette technique de surcharge des pointeurs de souris. On pourrait en effet imaginer de la même façon que le pointeur puisse rendre compte d'une activité de jeu telle qu'un clic sur un bouton ou bien un glissé-déposé par un changement d'icône. Néanmoins, nous doutons de la pertinence de cette fonctionnalité à l'heure actuelle dans la mesure où les valeurs de jeu ne sont pas partagées entre les utilisateurs. Ils verraient alors par exemple que quelqu'un manipule un potentiomètre, mais ne pourraient pas observer le changement de valeur. Notons pour finir que si ce genre de solution graphique paraît attrayante à mettre en place, et pourrait bénéficier à l'utilisateur en lui fournissant un meilleur aperçu de l'activité des autres au sein du patch, en revanche, elle peut aussi être légèrement envahissante au sein de l'interface du patch et gêner sa lecture. Aussi nous faudra-t-il prévoir, si nous la mettons en œuvre, la possibilité de configurer ou de désactiver cette option au sein de l'interface graphique du logiciel.

8.2.2. Vue en radar

Une autre information qui pourrait être intéressante à partager aux autres utilisateurs pour améliorer la conscience de groupe au sein du patch est celle de la zone visualisée par chaque utilisateur. *Où se situe tel utilisateur au sein du patch ? Comment voir la partie visualisée par tel utilisateur ?* Ou encore, *comment inviter un collaborateur à visualiser la partie que je suis en train de visualiser au sein d'un document ?* L'interface graphique du logiciel pourrait fournir des fonctionnalités qui permettent aux utilisateurs de répondre de manière pratique à ces questions et ces besoins. Ces informations et ces fonctionnalités peuvent notamment être traduites et mises en place graphiquement par l'ajout d'une vue secondaire représentant de manière rétrécie l'ensemble du document. Ce principe de vue globale réduite, de carte, ou encore vue en radar (*radar view*) comme l'appelle C. Gutwin [Gutwin, 1997], est présent dans de nombreuses applications, pas forcément collaboratives ni relatives au domaine musical.

L'éditeur de texte *SublimeText*[219] propose notamment d'afficher une vue globale du texte, appelée *minimap*, à côté de la vue standard, qui sert alors à mieux se repérer au sein d'un document conséquent. La station de travail audionumérique *Pro Tools*[220] propose, elle aussi, depuis longtemps une représentation de la session multipiste globale dans son interface.

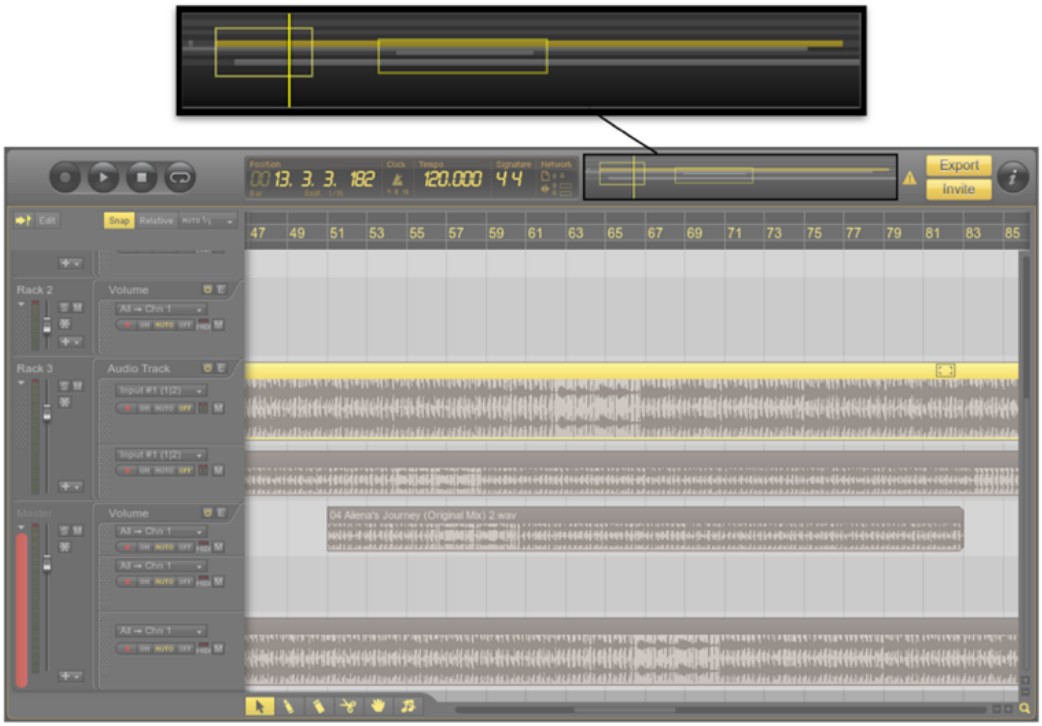

Figure 74 – Capture d'écran d'une session multipiste au sein de l'interface du logiciel *OhmStudio* dans laquelle deux personnes sont connectées. Le composant du dessus est une vue grossie de la fonctionnalité que nous appelons *minimap* dans laquelle il est possible d'obtenir une vue d'ensemble de la session, l'endroit où l'on se trouve et la zone que sont en train de visualiser les autres utilisateurs connectés.

Ce principe de vue en radar est aussi disponible au sein de l'interface de l'application collaborative *OhmStudio*, où il est alors possible de visualiser à la fois son propre emplacement au sein de la session multipiste mais aussi de repérer, localiser et suivre les autres personnes

[219] https://www.sublimetext.com/ [consulté le 20/09/18].

[220] https://www.avid.com/fr/pro-tools

connectées à cette même session en temps réel [Figure 74]. L'interface agit aussi comme une barre de défilement alternative permettant de naviguer plus facilement au sein de la session.

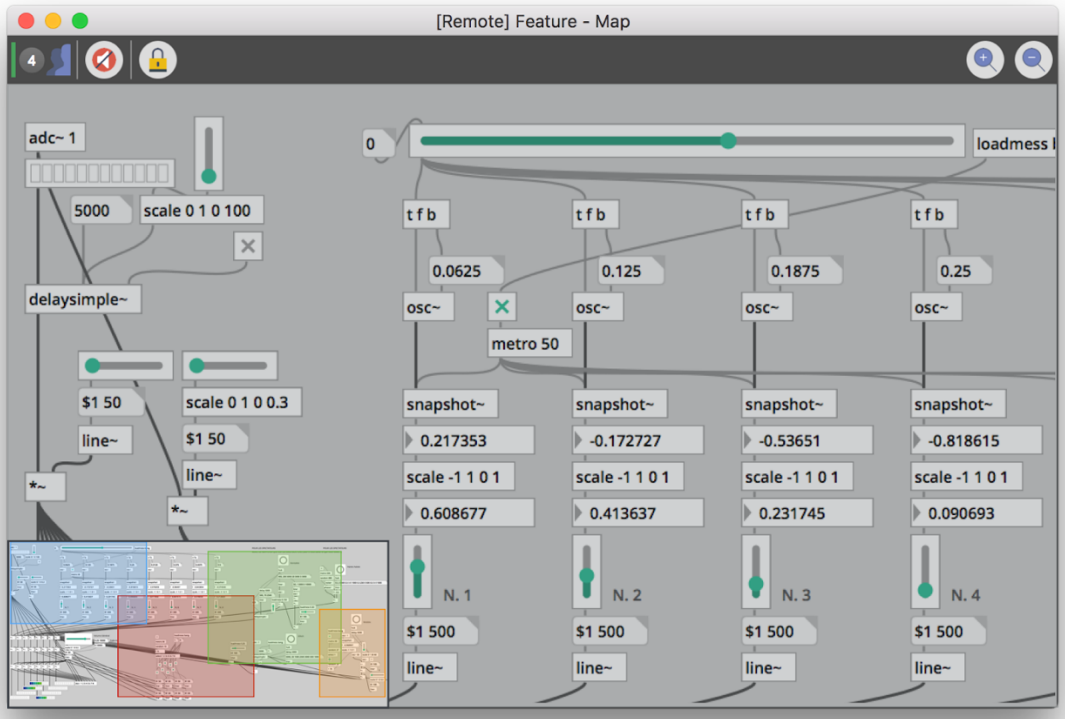

Figure 75 – Maquette d'interface graphique présentant l'ajout d'une fonctionnalité de *vue en radar* au patch, pour l'instant non-présente dans Kiwi mais qui pourrait faire l'objet d'une mise en œuvre potentielle dans une future version de l'application. Quatre personnes sont connectées au même patch. La fenêtre présente la vue locale d'un utilisateur de ce patch. Le cadre en bas à gauche offre une vue globale réduite de l'ensemble du patch au sein duquel les rectangles de couleur servent à représenter la zone de vue courante des autres utilisateurs. La zone de vue de l'utilisateur local est représentée par le rectangle bleu.

L'adaptation de ce principe à l'application Kiwi pourrait ressembler graphiquement à ce qui est proposé à la [Figure 75]. Chaque utilisateur aurait la possibilité d'afficher cette *vue en radar* pour se repérer plus facilement au sein du patch et visualiser l'emplacement des autres participants. On pourrait aussi imaginer que cette vue soit interactive et qu'elle permette de se déplacer au sein du patch ou d'adapter la vue locale pour qu'elle corresponde à ce que visualise un autre collaborateur (*e.g.* taille de la fenêtre, grossissement ou rétrécissement de la zone de

vue). Ce type de fonctionnalité serait alors utile pour coordonner à la fois l'activité d'édition mais aussi celle de jeu au sein du patch.

Dans les deux dernières sous-sections, nous avons étudié des solutions qui pourraient être mises en place dans le cadre de Kiwi pour améliorer la conscience de groupe au sein de l'application dans des situations où l'interaction se fait essentiellement de manière synchrone. Dans la partie suivante, nous étudierons la problématique de la conscience de groupe appliquée aux interactions asynchrones au sein des documents.

8.2.3. Conscience du changement

La collaboration asynchrone est rendue possible par la persistance des patchs créés et stockés en ligne. Les documents peuvent être édités à distance, à différents moments, par la même personne ou par des personnes différentes. Si un utilisateur reprend un travail qu'il avait lui-même entamé il y a un certain temps sur un document, il ne se souvient pas forcément des étapes ou des différents choix qu'il a dû effectuer pour produire la version courante. De la même façon, si un utilisateur se connecte à un document qu'il n'avait jamais consulté auparavant, il peut avoir envie de connaître le processus cognitif qui a mené à l'élaboration d'un traitement. Lorsque plusieurs utilisateurs collaborent au sein d'un même patch, ils disposent tous de la dernière version du document néanmoins, tous n'ont pas une vision globale des modifications apportées au document dans la mesure où ils ne les ont pas vu apparaître au moment où elles ont été réalisées par d'autres. Pour coordonner leur travail, les utilisateurs ont donc besoin que l'interface du collecticiel puisse traduire et témoigner des changements opérés sur un document dans le temps, fournir des moyens qui permettent aux utilisateurs d'avoir conscience des changements [McCaffrey, 1998]. Nous avons malheureusement manqué de temps au sein du projet MUSICOLL pour nous confronter à cette problématique et y apporter des réponses satisfaisantes. Les enjeux relatifs à la conscience du changement sont néanmoins nombreux,

nous tenterons donc d'en donner ici un aperçu à titre prospectif afin d'envisager des perspectives potentielles au projet. Il s'agit notamment de déterminer de quelles informations de changement les utilisateurs ont réellement besoin (8.2.3.1), de trouver les moyens techniques de capturer et de stocker ces informations de manière persistante (8.2.3.2), mais aussi de les afficher au sein de l'interface graphique afin qu'elles soient réellement intelligibles et pertinentes (8.2.3.3).

8.2.3.1. Quels changements ?

La première question que nous devrons nous poser est de savoir quelles informations sont pertinentes pour l'utilisateur afin qu'il puisse avoir conscience des différents changements opérés par lui ou par les autres au sein d'un document. L. McCaffrey propose une liste de questions que les utilisateurs peuvent se poser à propos des changements opérés sur un document partagé [McCaffrey, 1998, p. 6-8] :

- *Quels* sont les changements ?
- *Où* ces changements se sont-ils produits ?
- *Qui* a effectué ces changements ?
- *Comment* l'utilisateur a effectué ces changements ?
- *Quand* ces changements ont-ils eu lieu ?
- *Pourquoi* ces changements ont été faits ?

Il est important de définir aussi selon nous le niveau de granularité souhaitée au niveau de ces informations de changement. Dans certains cas on cherchera à obtenir des informations générales sur les changements apportés à un document dans son ensemble. Qui a modifié le document et quand ? Qu'est-ce-que les modifications apportent au document d'un point de vue fonctionnel ? Pourquoi et dans quel but avoir fait ces changements ? La réponse qu'un

utilisateur attendrait de la part du système serait, par exemple, que le document a été modifié par un utilisateur nommé « fred », hier, et qu'un paramètre de raréfaction au module de granulation a été ajouté au patch. Dans d'autres contextes on pourra aussi se poser le même type de questions mais à un niveau de détail plus fin, par exemple, qui a créé un objet spécifique au sein du patch ou encore, au sein d'une boîte commentaire, qui a modifié le texte, à quel moment, que contenait l'objet avant que la modification n'ait lieu. Ce besoin spécifique a d'ailleurs été exprimé lors des retours obtenus de la part des étudiants de l'expérience du cours de Kiwi donné à l'Université et relaté dans la section précédente.

Selon le type d'information demandée, la manière de les capturer, de les stocker ou encore de les afficher au sein du logiciel pourra aussi différer.

8.2.3.2. Capture et stockage des informations

Comment capturer et stocker les informations de changements ? Peut-on les obtenir de manière automatique, ou doit-on demander à l'utilisateur de fournir lui-même des clefs de compréhension aux autres ? Tout dépend selon nous du type d'information demandée. Les circonstances du changement liées aux questions du *Quel*, *Où*, *Qui, Quand* peuvent être fournies de manière automatique par le système dans la mesure où il dispose de ces informations. La réponse aux questions circonstancielles du *Comment* et du *Pourquoi* seront en revanche plus difficilement automatisables dans la mesure où elles font appel à la sémantique de l'action et à l'intention du créateur qui sont souvent des notions subjectives.

Les opérations réalisables liées à l'activité de conception du patch sont de différentes natures. Des objets peuvent être créés, édités, renommés, redimensionnés, déplacés, ou encore reliés entre eux par des liens. Ces actions représentent donc les changements sur un document, observables de manière automatique par le système. Les opérations réalisées par un utilisateur

sur le document sont actuellement stockées localement au sein d'un historique durant la phase de conception du patch. Ce mécanisme est à la base du système d'*undo/redo*[221], qui permet dans un certain sens de fournir des informations sur des changements opérés précédemment sur un document de manière automatisée. Par exemple, lorsqu'un utilisateur ajoute un objet *slider* au patch ou créé un lien entre deux objets, ces modifications sont décrites de manière automatique par l'application et affichées sous la forme de messages simples tels que « *slider object added* » ou encore « *link added* ». Néanmoins, l'inadaptation de ce système à la problématique de la conscience du changement repose sur plusieurs éléments. Premièrement il n'est pas persistant, ce qui veut dire que les informations sont éphémères et ne sont plus présentes lors d'une réouverture du document. Deuxièmement, il ne laisse voir que la dernière action réalisée et ne prend en considération que celles réalisées par l'utilisateur local. On ne peut donc pas voir quelle est la dernière action à avoir été effectuée sur le document de manière globale et encore moins les précédentes.

Une première solution technique envisageable pour gérer un historique des modifications globales pourrait être de nous appuyer sur la bibliothèque *flip* pour stocker chaque transaction arrivant au serveur au sein d'un historique global persistant qui permettrait ainsi de disposer de toutes les modifications effectuées par l'ensemble des utilisateurs au sein du patch depuis sa création. Cet historique pourrait alors être requêté par les applications clientes qui pourraient ensuite représenter visuellement ces transactions et leur métadonnées associées (*e.g.* description de la modification, heure et auteur de la modification). Cette première approche, si tant est qu'elle soit réalisable sous cette forme, ne nous convient pas entièrement. Les messages seraient à notre avis en trop grande quantité pour être utiles et intelligibles pour l'utilisateur. Même en

[221] Nous avons décrit ce système dans la seconde partie de ce mémoire, voir sous-section 5.1.4 (p. 178).

améliorant la description des transactions, ces informations resteraient trop génériques et surtout à un niveau de détail qui ne permettrait pas aux utilisateurs de prendre réellement conscience des changements globaux liés à l'édition d'un patch. Il pourrait donc convenir pour obtenir des informations à un niveau très détaillé mais pas pour décrire des changements plus généraux. Dans un système temps réel tel qu'il est mis en place dans Kiwi, chaque modification apportée au document produit une nouvelle version du document dans la mesure où chaque transaction est transmise directement au serveur. Pour améliorer la représentation des changements au sein du patch il nous faudrait stocker des états successifs du patch. Pour cela, on pourrait grouper certaines des modifications entre elles de manière automatique. Ce regroupement pourrait se faire selon un critère temporel – par exemple toutes les modifications apportées au document groupées par tranche horaire – ou encore par auteur – toutes les modifications effectuées par un utilisateur sur le document, quitte à perdre le détail spécifique de chaque opération. On pourrait alors générer un historique du document plus compréhensible. Néanmoins, ce mécanisme ne suffirait toujours pas à traduire de manière plus générale l'intention de l'utilisateur au moment où il a réalisé une modification sur le patch. Dans des systèmes de gestion de version tels que *git*, c'est l'utilisateur qui décide quand transmettre les changements. Une nouvelle version du document comporte alors généralement plus d'une modification et le système propose à l'utilisateur de les décrire sous forme textuelle. Notre intuition serait donc, pour offrir un aperçu plus pertinent des changements aux autres, de faire en sorte que ce soit aussi aux utilisateurs de décrire les modifications qu'ils ont apportées au document quand ils le décident. Un utilisateur pourrait alors travailler sur un patch, faire les modifications qu'il souhaite puis, lorsqu'il considère que les modifications qu'il a effectuées représentent un changement notable qui ait, de son point de vue, du sens au sein du patch, décrire textuellement ces changements (*e.g.* « *ajout d'un paramètre de raréfaction au module de granulation* »). Le serveur se chargerait alors d'enregistrer l'état du document au moment

où l'utilisateur l'a décidé avec des métadonnées associées telles que l'auteur, la date et la description sémantique du changement. Cette approche est aussi celle adoptée par des solutions professionnelles telles que *Google Docs* qui propose notamment de nommer les versions des documents au sein de son interface.

8.2.3.3. Représentation des changements

Dans la première partie de ce document, lors de l'étude des solutions existantes en matière de patching collaboratif, nous en avons évoqué deux qui permettaient de supporter l'activité de conception asynchrone de patch : celle du forum, avec la pratique du *Hippie Patching* et celle des gestionnaires de version de type *git*[222]. Si nous avons montré que ces outils ne convenaient pas à une édition collaborative resserrée au sein d'un patch, comme elle peut désormais être mise en œuvre au sein de Kiwi, nous avons vu aussi qu'ils proposaient différentes fonctionnalités qui permettaient aux utilisateurs de prendre conscience des changements opérés sur les documents. Dans le cas du forum, les messages postés créaient par eux-mêmes une trace persistante de l'activité globale. Les utilisateurs avaient aussi mis en place un mécanisme qui permettait de visualiser les contributions de chacun passant par une coloration manuelle des objets au sein du patch [Figure 5, p. 44]. Les gestionnaires de versions permettent quant à eux d'offrir une représentation graphique de l'historique des modifications qui peut être *décoré* de plusieurs métadonnées telles que l'auteur des modifications, une description des modifications sous forme textuelle ajoutée par son auteur, ou encore une date de modification [Figure 8, p. 60]. Ces derniers offrent aussi un niveau de détail plus fin grâce aux *diffs* qui représentent les changements effectués entre deux versions d'un document. Ces solutions ne sont néanmoins pas dédiées spécifiquement à la pratique du patching. Aussi, les traces de l'activité ne sont pas toujours automatisées, elles sont souvent aussi détachées du document et la représentation des

[222] Voir sous-sections 2.1.1 (p. 36) et 2.1.2 (p. 55) de ce document.

changements n'est pas toujours adaptée. Dans la mesure où nous disposons désormais de notre propre solution logicielle, il nous est désormais possible d'envisager de nouvelles solutions qui permettront de répondre de manière plus spécifiques aux enjeux liés à la conscience du changement appliqués à la pratique du patching audio.

Pour répondre aux questions *Quel*, *Où*, *Qui, Quand*, de manière granulaire, on pourrait alors imaginer reprendre la solution mise en place par les utilisateurs du forum mais la supporter dans le cadre de Kiwi de manière automatique en offrant, par exemple, un calque supplémentaire à l'interface du Patcher qui permettrait de pointer la contribution de chaque utilisateur au sein du patch, ou encore qui proposerait d'autres métadonnées contextuelles sur l'historique des modifications apportées sur chaque objet.

La représentation globale des modifications apportées sur un document pourrait prendre la forme d'une interface dédiée qui pourrait fournir des informations similaires à celles proposées dans les solutions évoquées, mais qui auraient l'avantage d'être ici intégrées à l'application et donc d'être directement liées aux documents. On pourrait alors imaginer une solution qui permettrait de créer un instantané du document à un moment précis, nommer et décrire textuellement une version spécifique, pour pouvoir ensuite visualiser l'historique de ces différentes versions à côté du patch. Pour se représenter les modifications apportées entre deux versions du même document, les utilisateurs pourrait alors se référer aux informations textuelles fournies par les autres utilisateurs, mais on pourrait aussi imaginer d'autres types de représentations. Lors de l'étude des logiciels de gestion de versions nous avions donné un exemple de *diff* d'un fichier Pure Data et Max [Figure 9, p. 60].

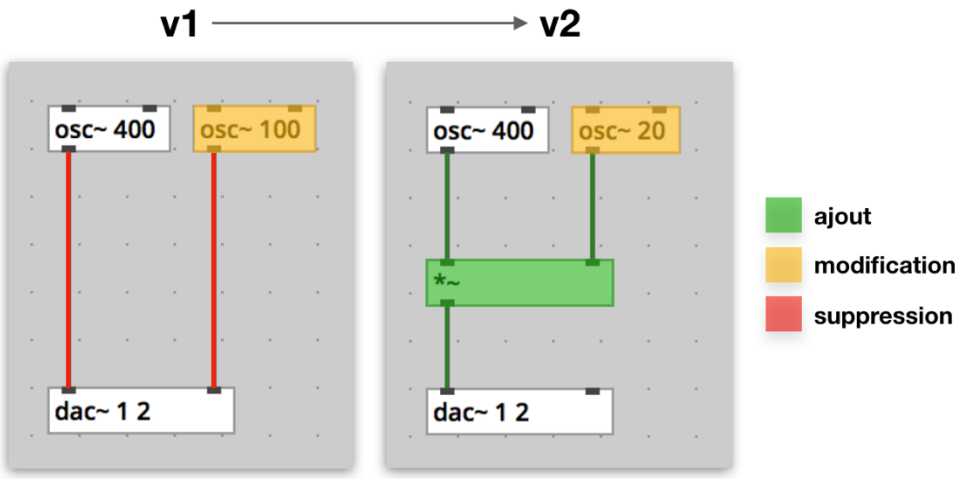

Figure 76 – Vue maquettée d'une représentation possible d'un *diff* entre deux versions différentes d'un même patch Kiwi. Les éléments ajoutés entre l'une et l'autre version apparaissent en vert, les modifications apportées aux objets en jaune, enfin, les suppressions sont représentées en rouge.

Nous avions aussi soutenu à cette occasion, que la représentation était peu adaptée, dans la mesure où les changements représentés étaient ceux liés à la représentation textuelle du document, et pas à la représentation graphique d'un patch à laquelle l'utilisateur est habitué. Une solution envisageable serait alors de repenser ce système de *diff* pour l'adapter plus spécifiquement à la pratique du patching [Figure 76]. Les utilisateurs pourraient alors voir quels objets ou liens ont été modifiés, ajoutés ou supprimés entre deux versions d'un patch, directement sur l'interface graphique du patch et à travers une représentation qui leurs serait plus familière.

Les problématiques liées à la conscience du changement n'ont pas pu être traitées dans l'intervalle temporel dédié à la recherche et au développement de l'application Kiwi. Bien que les utilisateurs aient trouvé des moyens détournés de représenter des changements ou laisser des traces de leurs activités aux autres au sein de l'espace de travail partagée, notamment via la mise en place d'objets *commentaires*, ces solutions actuelles ne sont pas optimales et pas automatisées. Les solutions évoquées dans ce cadre, aussi bien au niveau de la capture, du

stockage de l'information de changement mais aussi de sa représentation ne sont à prendre que d'un point de vue prospectif. Ce sont des maquettes, des premières pistes de réflexions et de recherches qui seraient selon nous intéressantes à poursuivre. Aussi, l'une des perspectives du projet se situe dans l'apport de nouvelles solutions concrètes qui viendrons soutenir les interactions liées à l'activité asynchrone de conception de patch.

Dans cet ordre d'idées, l'autre enjeu de la collaboration asynchrone se situe aussi dans la manière d'archiver les documents ; de faciliter l'accès aux documents dans le temps, ou encore de pouvoir contrôler qui y a accès. Ces problématiques, qui sont relatives à la gestion des sessions, seront abordées dans la section qui suit.

8.3. Gestion des sessions

La gestion des sessions est un aspect fondamental du collecticiel dans la mesure où elle permet aux utilisateurs de se retrouver virtuellement et de contrôler la manière dont ils sont interconnectés à travers l'application aux différents espaces de travail, c'est-à-dire dans notre cas aux différents patchs depuis le serveur. Comme la plupart des développements qui sont intervenus dans le cadre de Kiwi, celui-ci est issu d'un processus itératif. Au début nous ne pouvions travailler qu'en local sur nos machines, il n'y avait alors aucune session à *gérer* dans la mesure où les patchs que nous manipulions n'étaient connectés à aucun serveur. Le premier développement que nous avons entrepris, allant dans le sens d'une conception collaborative de patch, a permis à plusieurs personnes de se connecter à un même document en même temps. Nous avions alors qu'une seule session, qui était éphémère, dans le sens où le document n'était pas sauvé sur le disque. Le développement qui a suivi a permis de gérer plusieurs sessions persistantes, en les stockant de manière décentralisée sur chacune des machines clientes qui

tenaient alors aussi le rôle de serveur de documents[223]. À cette occasion nous avons introduit une première version du composant *Document Browser* [Figure 51, p. 194], qui permettait de créer de nouveaux documents sur chacun des serveurs présents sur un réseau local, mais des problèmes d'accès à ces documents se sont alors posés. Le passage à une architecture centralisée en ligne[224] nous a ensuite permis d'offrir un accès facilité aux documents distants stockés sur le serveur, de mettre en place un système d'authentification, d'ajouter des métadonnées associées aux documents, telles qu'un nom, une date de création, le nom de la personne ayant créé le document ou encore de celle qui l'a ouvert en dernier. Nous avons alors mis à jour l'interface graphique dédiée à la gestion locale des documents distants présents sur le serveur pour exposer ces nouvelles informations et fonctionnalités. Cette interface permet aujourd'hui de lister l'ensemble des documents présents sur le serveur, de les ouvrir, les renommer, les télécharger localement ou encore d'en téléverser de nouveaux sur le serveur. Néanmoins, cette interface, mais aussi plus généralement la manière dont sont gérées les sessions au sein de l'application restent perfectibles. Dans la première sous-section, nous évoquerons quelques améliorations qui sont envisageables à court terme au niveau du composant *Document Browser* (8.3.1). Dans les deux sous-sections suivantes nous évoquerons des problématiques plus complexes, que nous n'avons pas eu le temps de traiter dans le cadre du projet MUSICOLL, à savoir celles relatives au contrôle d'accès aux documents et à la création d'un espace personnel (8.3.2) puis celles relatives à la gestion d'un mode hors-connexion (8.3.3).

[223] Le fonctionnement de ce système est décrit en détail à la sous-section 5.2.1 (p. 192) de ce document.

[224] Voir sous-section 5.2.2 (p. 196) de ce document.

8.3.1. Amélioration de l'interface de gestion de sessions

Les documents visualisables au sein de la fenêtre *Document Browser* correspondent à l'ensemble des documents stockés en ligne sur le serveur. À chaque fois que quelqu'un crée un nouveau document, celui-ci est accessible automatiquement à tous les utilisateurs qui le voient alors apparaître comme une nouvelle entrée dans la liste [Figure 56, p. 206]. Cette interface permet donc à plusieurs personnes de créer leurs propres sessions de travail et de s'y connecter facilement, ce qui était un des enjeux principaux relatifs à la gestion des sessions que nous avons définis dans la première partie de ce mémoire. Néanmoins, au cours du projet et des différents cas d'utilisation du logiciel nous nous sommes aperçus que de petites améliorations seraient profitables pour pouvoir mieux s'y retrouver au sein de l'interface. Certaines de ces modifications nous ont d'ailleurs été suggérées par l'équipe éducative et les étudiants lors des retours obtenus suite à l'expérience pédagogique du cours pilote sur Kiwi. Plusieurs développements sont envisageables à court terme pour pouvoir améliorer la manière de retrouver facilement un document à partir de l'interface. On pourrait par exemple envisager l'ajout d'une barre de recherche ou encore d'un système d'étiquette à apposer sur les documents afin de les identifier plus rapidement grâce à un mécanisme de filtrage qui s'ajouterait à celui de tri actuellement en place. Dans cet ordre d'idées, l'ajout de nouvelles métadonnées associées aux documents, permettant de décrire notamment la fonctionnalité d'un patch ou permettant d'apposer des commentaires relatifs au traitement qui y est opéré, pourrait aussi être bénéfique. Une autre possibilité serait aussi de gérer une arborescence au sein de l'interface avec un système de groupe qui permettrait de *ranger* les patchs dans des dossiers afin de mieux organiser l'espace public. Cela demanderait à la fois un développement côté serveur pour pouvoir gérer de nouvelles métadonnées relatives aux documents mais aussi du point de vue de l'interface graphique pour pouvoir refléter ces changements et offrir ces nouvelles fonctionnalités à l'utilisateur du logiciel. Néanmoins, l'enjeu principal relatif à la gestion de

sessions se situe à plus long terme dans le fait de supporter un espace personnel qui permette aux utilisateurs d'organiser comme ils le souhaitent leurs documents et de contrôler qui peut y avoir accès.

8.3.2. Vers un espace personnel et un contrôle d'accès aux documents

Comme nous avons pu le voir, les utilisateurs au sein de Kiwi ne disposent actuellement que d'un espace en ligne public pour y stocker leurs patchs. Pour accéder à cet espace, l'utilisateur doit obligatoirement être authentifié auprès du serveur, mais chaque utilisateur correctement authentifié a alors accès à l'ensemble des documents présents sur le serveur et dispose des mêmes droits au sein de l'interface du logiciel. Il peut renommer tous les documents et effectuer les modifications qu'il souhaite au sein des différents documents. Le point positif est que cela permet un accès rapide à la collaboration dans la mesure où tous les documents sont automatiquement accessibles par tous les utilisateurs du logiciel. Le revers est que tout repose actuellement sur la bienséance et le respect mutuel des utilisateurs. Rien empêche donc un utilisateur de renommer des documents ou de supprimer leur contenu sans l'accord des autres. Si ce comportement largement permissif de l'application peut être toléré dans le cadre d'une utilisation expérimentale ou contrôlée du dispositif[225] , il l'est beaucoup moins dans d'autres contextes. Pour remédier à ça il devient donc nécessaire d'introduire la notion de rôle au sein de l'application mais aussi de droits pour permettre aux utilisateurs de contrôler l'accès aux différents documents sur lesquels ils sont amenés à travailler.

Il est tout à fait envisageable de créer un espace personnel privé qui puisse n'être accessible que par un seul utilisateur. Nous disposons d'ailleurs déjà actuellement d'un système

[225] En utilisant, par exemple, son propre serveur de document.

d'authentification et d'autorisation qui permet de contrôler et restreindre l'accès à certains documents même si nous ne l'avons pour l'instant pas exploité[226]. L'utilisateur pourrait alors stocker ses patchs sur un espace en ligne qui lui serait dédié et être sûr que personne ne vienne modifier les documents qu'il y entrepose. En revanche pour gérer ensuite la question du partage de document, d'autres questions émergent rapidement, notamment du point de vue de la représentation de ce système au sein de l'interface graphique. Comment faire en sorte qu'un utilisateur puisse partager un document spécifique avec un autre, comment gérer les droits d'accès partagés aux documents ? Le développement d'une solution allant dans ce sens se dirigerait alors vers la mise en place d'un système proche d'un réseau social où des personnes pourrait, par exemple, suivre d'autres personnes au sein de l'application, les inviter à collaborer au sein d'un document spécifique, voir les documents qu'ils partagent, ou encore être notifiés des actions relatives à la gestion de ces documents par les autres.

Se pose aussi, dans la problématique plus générale du contrôle d'accès, la question des rôles et des permissions. Nous pourrions, pour chaque document, associer des rôles spécifiques à chaque personne qui pourrait alors avoir le droit d'effectuer ou non certaines actions sur un document en fonction des permissions qui lui sont accordées. On pourrait alors imaginer supporter des rôles de propriétaire, d'administrateur, d'éditeur, de commentateur ou encore de spectateur. Le propriétaire pourrait par exemple effectuer l'ensemble des actions, l'administrateur aurait le droit d'inviter d'autres personnes sur le document, l'éditeur pourrait éditer le patch mais pas inviter quelqu'un d'autre, le commentateur aurait simplement le droit

[226] Ce système repose sur un jeton de sécurité (JWT) qui est déjà mis en œuvre au sein de Kiwi, et dans lequel il est possible d'intégrer facilement des notions de rôle et de droits d'accès. Voir sous-section 5.2.2 (p. 196) de ce document.

d'ajouter des commentaires au patch, enfin un spectateur pourrait observer la conception d'un traitement mais ne pourrait par exemple pas intervenir dans le processus de création.

La mise en œuvre relative à la problématique du contrôle d'accès au sein de Kiwi n'est pas triviale. Cela demanderait de faire des modifications à la fois du côté du serveur, pour organiser et orchestrer ce genre de fonctionnalités, mais aussi du côté de l'interface graphique du logiciel pour permettre aux utilisateurs d'effectuer l'ensemble de ces actions mais aussi pour en restreindre certaines. Cet enjeu est néanmoins primordial si nous voulons que Kiwi puisse être utilisé à plus grande échelle. Nous l'envisageons donc comme une perspective de développement à moyen ou long terme incontournable, mais qui devrait encore faire l'objet de spécifications plus détaillées afin de savoir réellement comment elle doit être appréhendée.

8.3.3. Gestion d'un mode hors-connexion

Un autre aspect relatif à la gestion des sessions que nous n'avons pas pu traiter comme nous l'aurions voulu au sein de l'application est celui de la transition entre un mode d'édition en ligne et un mode d'édition hors ligne. Kiwi permet actuellement de travailler en local sur des patchs, comme on pourrait le faire sur des logiciels de patching traditionnels, ou bien sur des documents distant en se connectant au serveur. Les documents locaux peuvent être téléversés sur le serveur pour permettre à d'autres personnes d'y contribuer. À partir de la même interface, les documents distants peuvent être téléchargés depuis le serveur pour pouvoir être utilisés localement sans nécessiter une connexion à internet. Lorsqu'un utilisateur est en train de travailler sur un document distant et qu'il subit une déconnexion, le logiciel propose alors de continuer l'édition du document en mode hors-ligne, comme nous l'avons illustré à la [Figure 58, p. 211]. En revanche, aucune solution n'est pour l'instant proposée à l'utilisateur au sein de l'application pour lui permettre de reprendre une édition en ligne d'un document suite à une déconnexion. En l'état, les utilisateurs se retrouvent avec une version locale du document qui

est détachée de la version de référence présente sur le serveur. Ils peuvent continuer à travailler dessus, mais aucun moyen logiciel n'est mis à leur disposition pour leur permettre de rapatrier leurs modifications sur la version en ligne et ils ne peuvent pas non-plus bénéficier des modifications apportées parallèlement par d'autres personnes sur le document. Nous nous retrouvons alors paradoxalement dans une problématique proche de celle que nous souhaitions en fait régler par la création de Kiwi, à savoir celle relative à la gestion de l'édition collaborative d'un même document sur plusieurs sites distants. Pour gérer les cas de déconnexion temporaire, la solution pourrait être de stocker l'ensemble des transactions localement puis d'attendre qu'une reconnexion ait lieu pour transmettre les modifications au serveur qui pourrait alors les prendre en compte à ce moment-là et synchroniser à nouveau les deux versions du document. Cependant, plus le temps de déconnexion au serveur est conséquent, plus les risques de conflits deviennent importants du fait que plusieurs utilisateurs éditent le document sans que leurs modifications ne soient synchronisées. La solution à adopter dans ce cas serait proche de celle proposée par des outils de gestion de version traditionnels à savoir, d'effectuer lors d'une reconnexion au serveur une fusion des modifications locales avec la version en ligne du document. Ce mécanisme est pris en charge par l'API *flip* et ne devrait donc pas poser trop de problème de mise en œuvre. Quant à la représentation graphique des conflits, nous pourrions nous inspirer, ici aussi, du type de solution envisagée précédemment à la [Figure 76, p. 277]. La gestion du mode hors-connexion fait donc aussi partie des perspectives du projet et devrait faire l'objet d'un développement futur qui demanderait néanmoins d'être mieux spécifié avant d'être entrepris.

8.4. Jeu collaboratif

Les enjeux en matière de jeu collaboratif au sein de Kiwi sont nombreux et restent encore très ouverts dans la mesure où ce n'était pas la problématique principale à laquelle nous voulions

répondre à l'origine du développement de Kiwi. Lorsque nous avons démarré le projet, nous souhaitions que chacun puisse éditer un document mais garde aussi un contexte d'exécution qui lui soit local, à la fois au niveau de l'application en général mais aussi au niveau des différents patchs. Ce choix avait été fait pour permettre notamment à deux utilisateurs d'explorer les possibilités musicales offertes par un traitement sonore de manière isolée par la manipulation indépendante des paramètres du patch sur leur machine. Nous ne voulions donc pas que les réglages, notamment liés à l'audio, à la fréquence d'échantillonnage du système, mais aussi aux valeurs internes des objets au sein des patchs, soient forcément partagés entre les utilisateurs, comme c'était notamment le cas dans les solutions où le traitement est centralisé, telles que *Purée Data* ou encore *peerdata*[227]. Nous voulions, par exemple, pouvoir allumer l'audio de l'application sur notre propre machine sans pour autant que cela ait pour conséquence de l'activer chez les autres participants connectés aux mêmes documents ou à l'application. La modélisation du document et l'architecture que nous avons mises en place ont permis de garantir ce contexte d'exécution local[228]. Aussi est-il possible dans Kiwi d'exécuter un traitement localement sur deux machines différentes et d'avoir un rendu qui diffère dans la mesure où les valeurs internes des objets ne sont pas partagées entre les utilisateurs.

Cependant, au cours du projet, nous avons voulu apporter une réponse à la problématique du jeu collaboratif, notamment car l'équipe pédagogique le demandait [Galleron & al., 2018], mais aussi car elle pouvait ouvrir de nouvelles perspectives en matière de création musicale. Cette réponse a alors pris la forme d'un nouvel objet ajouté au langage, l'objet *hub*, qui permet désormais d'offrir un espace de communication de valeurs entre les participants d'une session. L'approche mise en place actuellement fonctionne dans la mesure où elle a pu être éprouvée

[227] Solutions étudiées dans la première partie de ce mémoire. Voir sous-section 2.2.1 (p. 66) de ce document.

[228] Voir section 4.2 (p. 126) de ce document.

dans plusieurs contextes pédagogiques ou musicaux. Néanmoins, elle souffre à l'heure actuelle de plusieurs problèmes que nous tenterons d'exposer dans ce cadre afin d'en prendre la mesure et d'envisager d'éventuelles solutions ou pistes de réflexion pour l'avenir.

Le partage des valeurs de jeu au sein des patchs de Kiwi n'est toujours pas automatique ni systématique. En revanche, il peut être mis en place de manière dynamique par les utilisateurs en programmant cette fonctionnalité à l'aide du langage graphique. En ce sens, l'approche proposée par Kiwi se rapproche actuellement de celle adoptée par la solution *netpd* qui permet aussi de *programmer* la synchronisation des valeurs de jeu à partir de Pure Data par la création d'*instruments* spécifiques[229]. Le partage des données, tel qu'il est réalisable actuellement à travers l'objet *hub*, entre dans l'espace de communication, dans la mesure où il permet de communiquer de l'information aux différents participants. Cependant, comme le note D. Salber, « le contenu sémantique de cette information concerne les acteurs communicants. Il est étranger au système qui se contente de servir de messager »[230]. Ceci a selon-nous deux conséquences qui sont à la fois positives et négatives. La première est que cela fait de cette approche une solution très flexible qui permet notamment aux utilisateurs de s'approprier cet espace pour créer des modalités de jeu qui leurs sont propres. La seconde, est que nous ne pouvons pas contrôler les informations qui transitent par l'objet, ni à quoi vont servir ces informations. Dans un système ouvert tel que Kiwi il faut donc aussi considérer l'impact que peut avoir ce type de communication du point de vue de la sécurité. Le premier point que nous aborderons dans cette section vise à mettre en lumière cette dualité (8.4.1). Nous proposerons ensuite dans les deux points suivants des pistes de recherche et de nouvelles solutions envisageables afin d'améliorer d'autres aspects liés à la problématique du jeu collaboratif au

[229] Voir 2.2.2.2 (p. 80) de ce document.

[230] [Salber, 1995], p. 19.

sein de Kiwi. Nous discuterons alors de l'intérêt d'ajouter de nouvelles interfaces dédiées plus spécifiquement à cette pratique (8.4.2) et de disposer d'un système de stockage persistant de données (8.4.3).

8.4.1. Du partage de données au partage de fonctionnalités

Le simple fait d'avoir ajouté l'objet *hub* au langage, et donc un outil de communication inter-utilisateur dynamique au sein du patch, a eu des conséquences que nous n'avions pas forcément anticipées. Les choses qui étaient alors garanties comme locales ne le sont désormais plus.

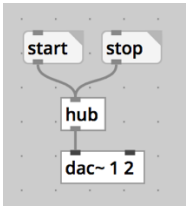

Figure 77 – Capture d'écran d'une partie de patch Kiwi permettant, en programmant cette fonctionnalité à l'aide du langage et de l'objet *hub*, d'allumer ou d'éteindre l'audio de l'application chez tous les utilisateurs connectés aux document.

Il devient, par exemple, tout à fait possible d'allumer ou d'éteindre l'audio de l'application Kiwi d'un autre utilisateur à distance [Figure 77], ou encore de poster des messages dans sa console, qui faisait normalement partie de l'espace de production privé. Cette fonctionnalité de partage est bien sûr très utile, car elle permet le contrôle collaboratif de patch et a pu être appréciée à ce titre dans les diverses représentations musicales évoquées au chapitre précédent. Grâce au fait de pouvoir poster des messages dans la console d'un autre participant nous avons aussi pu réaliser notamment un composant simple permettant la discussion instantanée [Figure 63, p. 222]. Néanmoins, l'objet *hub* peut aussi se révéler dangereux s'il n'est pas utilisé correctement, dans la mesure où il permet tout simplement de contrôler l'exécution d'un programme à distance – qui se présente dans Kiwi sous la forme d'un patch – sur une autre machine que la sienne. Aussi nous sommes-nous rendus compte, lorsque nous avons ajouté

l'objet *sf.record~* à l'application que nous y avions potentiellement aussi introduit une faille de sécurité. Nous avons choisi de l'exposer dans ce cadre pour des raisons didactiques. Comme le fait remarquer S. Greenberg, « Si l'application s'exécute dans l'espace de données d'un participant (*e.g.* au sein de son espace fichier), cette personne court le risque que ses fichiers soient intentionnellement ou par inadvertance endommagés par les autres participants à la conférence. »[231]. La figure présentée ci-dessous vise à illustrer cette problématique de manière concrète, appliquée à notre cas [Figure 78].

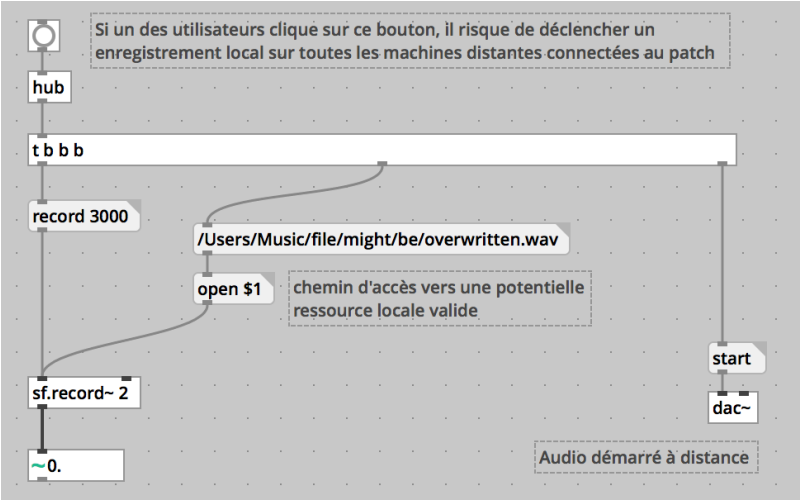

Figure 78 – Capture d'écran d'un patch Kiwi illustrant une utilisation de l'objet *hub* qui pourrait mener un utilisateur connecté à écraser le contenu d'un fichier son présent sur une machine distante connectée au même patch.

Lorsque l'on appuie sur le bouton présent en haut du patch, trois messages *bang* sont envoyés séquentiellement de droite à gauche par l'objet *trigger* « *t b b b* ». Le premier message a pour conséquence de démarrer le traitement audio de l'application, comme montré un peu plus haut [Figure 77] ; le second charge une ressource dans laquelle on peut écrire sur le disque en

[231] [Greenberg, 1990], p. 235, traduit de l'anglais : « If the application runs within the data space of one participant (ie within his file space), that person runs the risk of his files being intentionally or inadvertently damaged by the other conference attendees. »

spécifiant un chemin d'accès absolu vers un fichier ; le troisième déclenche l'enregistrement des données audio durant un laps de temps donné, égal ici à trois secondes. Ce patch présente donc jusqu'ici une procédure tout à fait standard d'enregistrement de fichier son. Mais le simple fait qu'un objet *hub* s'interpose entre le bouton et l'objet *trigger* change radicalement la portée de cette opération. En effet, si un des utilisateurs connectés au patch dispose des droits en écriture sur le fichier pointé par le chemin d'accès du message sur sa propre machine, un second utilisateur, connecté au même patch peut alors déclencher, à sa place, l'enregistrement d'un fichier son à distance, entraînant la création d'un nouveau fichier ou l'écrasement de son contenu dans le cas où le fichier existe déjà. Ce comportement peut parfaitement être programmé intentionnellement. On imagine un cas d'usage où un régisseur de studio fait tourner un patch Kiwi recevant et traitant de l'audio sur un poste relié à une carte son. Celui-ci pourrait alors décider d'ajouter un paramètre au patch qui lui permette de contrôler à distance l'enregistrement à partir d'une seconde machine connectée au même patch. Si cette fonctionnalité est tout à fait souhaitable dans ce cas, ou plus généralement dans le cadre d'une utilisation personnelle et réfléchie de Kiwi, ce type d'action est aussi potentiellement sensible dans un contexte collaboratif. En effet, l'utilisateur de l'application ne souhaite sûrement pas que quelqu'un d'autre puisse écraser n'importe quels fichiers sonores présents sur sa machine ou encore ne vienne enregistrer des heures d'audio sans qu'il ait donné son accord préalable sous le seul prétexte d'être connecté à un patch partagé ; ce qui peut malheureusement être ici le cas.

Notons que ce problème d'accès distant à une ressource n'est pas présent que dans Kiwi. Il existe en fait aussi dans toute solution qui permet de communiquer avec un logiciel ouvert tel que Max ou Pure Data. Si quelqu'un créé un *instrument* dans le système *netpd*, qui propose la même fonctionnalité que celle présentée ici, les utilisateurs seront exposés aux mêmes

problèmes, dans la mesure où il sera chargé et exécuté automatiquement sur toutes les machines connectées à la session de jeu. Si nous analysons les choses autrement, on se rend compte alors que ce n'est pas en soi le partage des données qui est en cause, mais le fait que ce partage de données puisse mener au partage d'une fonctionnalité dans le cadre spécifique des logiciels de patching. Pour M. Puckette « le développement de Pure Data a joué un rôle dans l'apparition d'une tendance qui risque de perdurer : on ne perçoit plus les logiciels comme des entités fixes qui opèrent sur des « *documents* » modifiables tels que des suites bureautiques sur des feuilles de calcul. La « *culture* » qui est en jeu s'incarne progressivement dans la fonctionnalité elle-même et de moins en moins dans le document. Les idées qu'on échange se retrouvent moins sous la forme de données passives et davantage dans la capacité de créer, modifier et utiliser les données de façon innovante » [Puckette, 2009, p. 189]. C'est précisément à cette dualité que nous touchons ici. Plus Kiwi s'agrémentera de nouvelles fonctionnalités, plus ce genre de failles de sécurité pourront aussi être nombreuses. Les environnements Max et Pure Data proposent des fonctions natives qui permettent notamment d'accéder au système de fichiers de la machine, ou encore de récupérer des périphériques d'entrées et de sorties tels que le clavier ou encore de la souris. Imaginons par exemple que nous ajoutions simplement un objet dans Kiwi donnant un accès aux fichiers sur le disque[232], et il serait alors possible de récupérer toute l'arborescence des fichiers présents sur une machine distante et peut-être même encore leur contenu, ce qui représente alors selon nous une faille de sécurité sérieuse à considérer. Aussi, un certain nombre d'objets embarqués de façon native dans la distribution de Max et Pure Data pourraient poser problème s'ils étaient mis en œuvre de manière iso-fonctionnelle dans le contexte d'un logiciel collaboratif tel que Kiwi.

[232] En envisageant par exemple un portage dans Kiwi des fonctionnalités présentes au sein des objets de type *filein* ou *filepath* tels qu'ils sont proposés dans l'environnement Max.

Nous n'avons malheureusement pas de solution satisfaisante à proposer pour l'instant. Bien qu'il ne soit d'habitude pas d'usage d'exposer un problème de sécurité de manière publique, tant qu'elle n'a pas été résolue au préalable, nous l'avons néanmoins fait dans ce cadre afin d'illustrer de manière plus concrète cette problématique que nous jugeons importante, et qui est intrinsèquement liée à celle du partage de l'exécution d'un patch. Aussi nous faudra-t-il être extrêmement vigilant quant aux prochaines fonctionnalités que nous choisirons d'offrir aux utilisateurs dans le cadre de Kiwi. Notons que ces problèmes pourraient être inexistants, comme ils l'ont d'ailleurs été jusqu'à présent, si tous les utilisateurs étaient bienveillants. Cet argument éthique ne tient néanmoins pas dans un système aussi ouvert que Kiwi dans son état actuel, où tout le monde a accès à l'ensemble des documents sur le serveur. Le fait de fournir un contrôle d'accès plus fin, tel que nous l'avons envisagé dans la sous-section précédente, pourrait donc aller aussi dans le sens d'une amélioration de la sécurité générale de l'application.

8.4.2. De nouvelles interfaces dédiées

Comme nous l'avons dit lors de la présentation de l'objet, le *hub* permet la communication de données entre les participants à une session mais pas la synchronisation automatique des valeurs. Cette synchronisation peut néanmoins être programmée grâce à un patron de conception commun aux environnements de patching traditionnels à l'aide d'un message de type « *set $1* » qui crée une boucle entre l'objet *hub* et l'interface graphique de contrôle. Nous l'avons exposé à la [Figure 61, p. 217] ou encore à la [Figure 68, p. 241]. Cette solution n'est néanmoins pas optimale dans la mesure où elle est loin d'être instinctive pour un utilisateur débutant au sein de l'environnement, comme nous avons pu le vérifier lors de l'expérience du cours pilote avec Kiwi. Les étudiants n'ayant pas compris le procédé avaient alors du mal à synchroniser leurs interfaces. Une approche visant à simplifier cette synchronisation pourrait donc être bénéfique afin d'offrir un accès plus direct au jeu collaboratif au sein du patch.

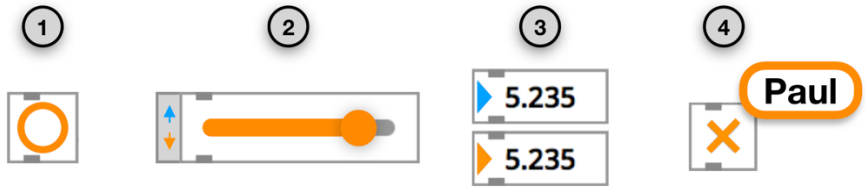

Figure 79 – Version maquettée d'interfaces de jeu collaboratives adaptée à partir des interfaces de contrôle actuellement présentes dans le logiciel Kiwi.

Aussi, une première idée pour améliorer l'utilisabilité du logiciel serait de créer de nouvelles interfaces de contrôle qui offriraient des fonctionnalités spécifiquement pensées pour le jeu collaboratif [Figure 79]. Ce type d'interface, que l'on pourrait préfixer du texte *kiwi* pour les différencier des autres (*e.g. kiwi.slider*), intègrerait alors des mécanismes qui garantiraient une synchronisation de leur(s) valeur(s) interne(s) entre tous les participants. Un utilisateur désireux d'ajouter une interface permettant le contrôle collaboratif d'un paramètre au sein d'un patch partagé pourrait alors simplement instancier l'une d'entre-elles et la relier directement aux objets destinés à être contrôlés, sans avoir à gérer manuellement sa synchronisation. Un développement allant dans ce sens permettrait aussi d'intégrer plus facilement des mécanismes de conscience de groupe spécifiques à l'activité de jeu au sein de ces interfaces. On pourrait alors imaginer que celles-ci puissent témoigner des valeurs transmises ou reçues de manière plus claire [Figure 79, vignette 2] ou encore que l'on puisse savoir qui vient de modifier la valeur [Figure 79, vignette 4]. Pour permettre de clarifier ce qui est de l'ordre du contrôle local et du contrôle collaboratif au sein d'un patch et ajouter plus de flexibilité entre ces deux modes, une idée serait aussi d'intégrer une option, par exemple au sein de la fenêtre du Patcher, qui permette à l'utilisateur d'activer ou de désactiver la synchronisation de ces interfaces. Un signe bleu au sein de l'interface de contrôle pourrait alors signifier que le paramètre est local et que sa valeur ne sera pas partagée, et un signe orange qu'il est collaboratif et que la valeur est synchronisée entre les participants [Figure 79, vignette 3].

8.4.3. Stockage de valeurs

Si un utilisateur trouve des réglages pertinents musicalement au cours de l'exécution d'un traitement sonore, il peut avoir envie de les partager avec quelqu'un d'autre momentanément ou de les sauver de manière persistante au sein du document pour que lui ou une autre personne puisse les rappeler ultérieurement. Kiwi ne dispose pour l'instant pas de système dédié au stockage des valeurs de jeu au sein d'un patch. La solution actuelle est donc de stocker chacune de ces valeurs au sein d'un boîte de *commentaire* pour s'en souvenir ou alors dans des objets *message* afin de pouvoir les invoquer plus tard lors de l'exécution du patch. Ces solutions permettent effectivement de stocker de manière persistante une valeur et de la partager avec quelqu'un au sein de l'espace de travail commun. Mais elles sont loin d'être satisfaisantes dans la mesure où il faut les mettre à jour manuellement.

On pourrait donc dans un premier temps imaginer une série de nouveaux objets qui permettraient de stocker différents types de valeurs de manière persistante au sein d'un document de manière plus automatique, comme un nombre flottant (*e.g. kiwi.float*), une chaîne de caractères (*e.g. kiwi.string*) ou une valeur générique (*e.g. kiwi.value*). La valeur interne de ces objets serait alors synchronisée entre tous les clients qui pourraient la mettre à jour ou la rappeler quand ils le souhaitent. Cela permettrait aussi aux nouveaux participants qui arrivent sur une session de disposer des mêmes informations que celles dont disposent déjà les autres participants connectés à la session, ce qui n'est actuellement pas réalisable avec le *hub*.

Nous pourrions aussi supporter dans Kiwi un système de stockage de préréglages plus complexe, analogue à celui de *preset* disponible au sein de l'environnement Max. Beaucoup de choses restent néanmoins à définir quant aux fonctionnalités à donner à tel système dans le contexte d'un patch partagé. Serait-il souhaitable de disposer d'un système de préréglages qui

soit global à tous les utilisateurs, spécifique à chacun, ou encore adaptable à ces deux types d'usage ? Ces trois cas sont en fait techniquement envisageables. Si nous voulons proposer un espace de stockage de préréglages commun à tous les utilisateurs, nous pourrons le faire en ajoutant un conteneur spécifiquement dédié à ça à la racine du document [Figure 21, p. 130, vignettes 1]. La modélisation actuelle du document nous permet aussi de disposer d'un espace de stockage spécifique à chaque utilisateur [Figure 21, p. 130, vignettes 1 et 4].

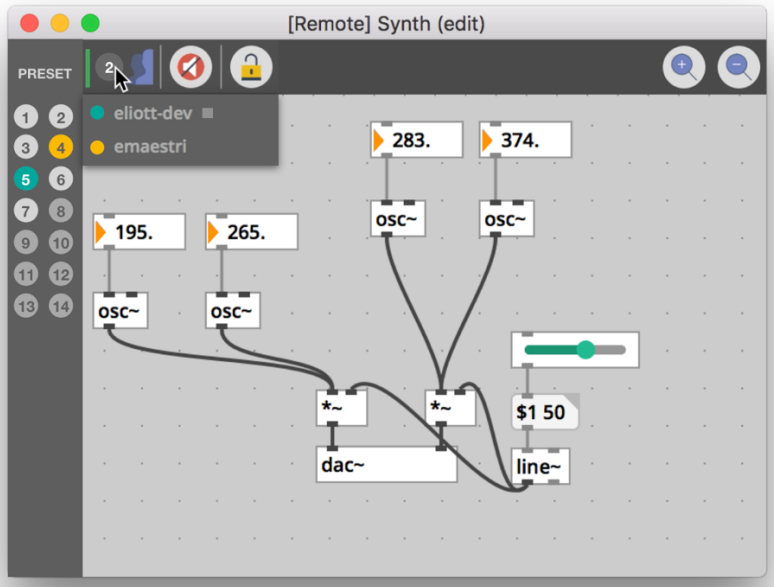

Figure 80 – Vue maquettée d'un système de préréglages intégré à la fenêtre Patcher de Kiwi.

Cet espace est actuellement utilisé uniquement pour stocker des informations relatives aux différentes vues des utilisateurs d'un patch comme le niveau de zoom, le mode d'interaction courant ou encore les sélections. Nous pourrions donc stocker de la même manière des variables de jeu qui soient spécifiques à chaque utilisateur. Chacun pourrait alors choisir de rappeler un de ses propres préréglages, un préréglage commun ou encore un des préréglages spécifiques d'un autre utilisateur. Une autre question est de savoir comment intégrer un système comme

celui-ci à l'interface graphique du logiciel et à la logique d'exécution du patch ? La figure ci-dessus expose une maquette d'interface graphique dans lequel il serait intégré directement à la fenêtre du patch [Figure 80]. Dans cette maquette, chaque utilisateur pourrait créer de nouveaux préréglages au sein d'un espace de stockage commun et rappeler les préréglages qu'il a créé ou que les autres ont créés, chacun peut voir aussi sur quel préréglage se situent actuellement les autres participants à la session. Néanmoins, les spécifications relatives à sa mise en œuvre effective restent encore totalement ouvertes. Devrait-on, par exemple, l'intégrer à la place, ou aussi, sous la forme d'un nouvel objet instanciable au sein du patch ? Par quel moyen les utilisateurs peuvent-ils choisir d'inclure ou non une valeur dans l'espace de stockage de valeur de jeu persistant ? L'enjeu sera donc, ici encore, d'étudier plus en profondeur ces questions et les besoins spécifiques des utilisateurs qui sont encore à mieux définir.

CONCLUSION GENERALE

Cette thèse propose une solution de patching audio adaptée à une approche multi-utilisateur.

Avant d'aborder la présentation de cette solution, il nous a fallu, dans la première partie de ce mémoire, dégager les principaux enjeux liés au sujet afin de définir les spécifications nécessaires à la création d'une application de ce type. Dans un premier temps, il était nécessaire de définir plus précisément ce qu'est un logiciel de patching et en quoi consiste la collaboration. Nous avons alors commencé par apporter une définition plus précise des environnements de patching audio en différenciant l'activité de conception de celle de jeu au sein du patch. Nous avons aussi présenté deux outils conceptuels. La matrice d'Ellis, qui permet de classer les types d'interaction selon un critère de temps et d'espace, et le trèfle des systèmes multi-utilisateurs qui nous a permis de classer les différents espaces d'un collecticiel, à savoir ceux liés aux fonctions de *production*, de *coordination* et de *communication*. Le premier chapitre a donc permis de définir des outils théoriques proposant une approche générale du sujet.

Dans le second chapitre, dans une perspective plus concrète, nous avons étudié une série de pratiques existantes et de solutions logicielles permettant aux utilisateurs de collaborer à partir des applications de patching audio traditionnelles. L'objectif était de faire ressortir la multiplicité des usages, les différents types d'interaction possibles, mais aussi d'exposer les différents enjeux musicaux, techniques et scientifiques, intrinsèquement liés à la dimension collaborative de ce genre de pratiques. Nous nous sommes pour cela appuyés sur la matrice d'Ellis et sur les deux types d'activités relatives au patching audio. Nous avons débuté par les solutions permettant une édition asynchrone de patch. Ces premiers exemples nous ont permis de montrer l'aspect ludique de la démarche mais aussi d'entrevoir certaines problématiques liées à la conscience de groupe ou encore aux conflits d'édition. Nous avons ensuite poursuivi

cette série d'études en analysant différents types d'outils conçus pour permettre l'édition et le jeu collaboratif synchrone, dans des situations où les personnes sont regroupées au même endroit ou dispersées géographiquement. Ces dernières nous ont permis de distinguer de nouvelles pratiques musicales, mais aussi d'identifier certaines limites aux choix techniques de mise en œuvre des solutions logicielles venant les soutenir.

À partir des outils conceptuels présentés dans le premier chapitre et des études réalisées dans le deuxième chapitre, il nous a été permis de définir, au sein du troisième chapitre, les enjeux plus spécifiques à l'application collaborative Kiwi que nous souhaitions développer. Nous avons alors annoncé et listé les principaux verrous scientifiques et techniques que nous avions à lever pour rendre, selon nous, la pratique du patching audio viable. Nous avons alors défini les caractéristiques des systèmes synchrones, la problématique liée au contrôle d'accès ou encore à la gestion des sessions. Il est aussi apparu que l'enjeu technique fondamental lié à la conception d'un système multi-utilisateur permettant une édition synchrone et distribuée de patch était de mettre en place un système permettant une réplique des données entre plusieurs sites. Ce type d'architecture s'est révélée être la condition initiale à la résolution des autres enjeux qui étaient de devoir fournir, par exemple, des éléments permettant une conscience de groupe, de réduire les problèmes de latence liés à des temps de réponse trop élevés mais aussi de fournir un contexte qui puisse être spécifique pour chaque utilisateur.

La deuxième partie de ce mémoire a été consacrée à la présentation du collecticiel Kiwi et à ses différents aspects de conception, dont les spécifications sont le résultat de la partie précédente. Cette partie a été guidée par la présentation des différents composants de l'interface graphique et divisée en trois chapitres qui reprennent la trame fournie par le trèfle des systèmes multi-utilisateurs.

Le quatrième chapitre, dédié à l'espace de production du collecticiel, nous a permis de dégager les principales fonctionnalités du logiciel. Nous avons alors détaillé les spécificités du langage graphique et fonctionnel de Kiwi et défini les différentes opérations relatives à la conception et à l'exécution d'un patch, dans un contexte à la fois mono et multi-utilisateur. Nous avons aussi pu distinguer ce qui relevait de l'espace personnel d'un utilisateur et ce qui relevait des données partagées. À travers l'étude de la modélisation du patch avec la bibliothèque *flip* et la description de l'architecture logicielle, nous avons montré comment nous pouvions garanti un contexte d'exécution local tout en permettant une édition collaborative du patch en temps réel.

Nous avons, par la suite, abordé l'espace de coordination dans le cinquième chapitre. Nous avons décrit notamment la manière dont l'édition concurrente du patch et les problèmes de conflits avaient été gérés. Nous avons détaillé les mécanismes de conscience de groupe mis en place pour permettre à plusieurs personnes de coordonner leurs activités au sein de l'espace de travail partagé. Par exemple, un utilisateur peut savoir qui est présent sur le patch ou encore les actions que sont en train d'effectuer les autres en observant leur sélection. Nous sommes ensuite revenus sur la gestion des documents partagés, qui était une condition pour permettre aussi l'édition asynchrone de patch. Nous avons alors décrit l'architecture réseau mise en place et l'interface graphique permettant à l'utilisateur de gérer ces documents depuis l'application.

Enfin, dans le sixième chapitre, nous avons abordé l'espace de communication. Nous avons alors exposé un objet spécifique de Kiwi, l'objet *hub*, qui permet une communication inter-utilisateur à travers le patch et ouvre donc la voie à des pratiques collaboratives de jeu. Les variables internes des objets ne sont pas partagées automatiquement entre les utilisateurs du patch, mais il est possible de les synchroniser grâce à cet objet pour qu'elles le soient.

La dernière partie de ce document était dédiée à la validation de la solution proposée, aux développements récents et aux perspectives futures.

Dans le septième chapitre nous sommes alors revenus sur différents cas d'utilisation du logiciel. L'enjeu était, dans un premier temps, d'éprouver la solution apportée en la confrontant à des conditions pédagogiques et musicales réelles afin de valider l'approche. Pour cela, le cours pilote donné avec Kiwi à l'Université Paris VIII a représenté un premier cadre d'étude idéal. Il nous a offert des situations d'interaction se situant dans les quatre quadrants de la matrice d'Ellis. Les étudiants ont pu éditer leurs patchs de manière collaborative en classe ou depuis chez eux, en temps réel ou de manière asynchrone. Cette expérience pédagogique singulière a aussi permis de valider l'hypothèse qu'un renouvellement de la pratique avait bien été induit par la solution logicielle proposée, notamment en favorisant un apprentissage par le faire-ensemble. Un renouvellement a d'ailleurs aussi pu être observé dans le contexte de la création musicale, notamment grâce à de nouvelles situations de jeu collaboratif à distance.

Si ces retours nous ont offert un terrain expérimental pour notre approche, ils ont aussi permis de contribuer de manière active au développement de l'application et notamment d'envisager de nouvelles pistes de recherche durant le projet. Dans le huitième et dernier chapitre de ce mémoire, nous avons cherché à exploiter certains de ces retours en proposant notamment de nouvelles solutions. Certaines de ces solutions, telles que l'objet *faust~*, qui permet d'éditer et d'exécuter du code Faust de manière collaborative, sont déjà en cours de développement et devraient être prochainement intégrées à l'application. D'autres enfin, sont envisageables à moyen ou plus long terme.

Au final, ce travail aura permis la réalisation du collecticiel de patching audio Kiwi, offrant un système adapté à la conception collaborative de traitements sonores. Il aura aussi fourni les outils théoriques et conceptuels ainsi que les bases logicielles permettant de projeter l'approche vers un jeu collaboratif.

Durant ces dernières années de recherche, consacrées en grande partie au développement de l'application Kiwi, nous nous sommes attachés à ce fruit, aussi, pour éviter de ne le voir dépérir, il nous faut pouvoir assurer sa pérennité au-delà du projet MUSICOLL. Cette pérennité passe selon nous par plusieurs choses. L'intérêt du logiciel se situant principalement dans sa dimension collaborative, il faut dans un premier temps s'assurer de maintenir un accès au serveur. Des solutions sont actuellement à l'étude avec l'Université Paris VIII, qui pourrait nous garantir un service d'hébergement afin que les utilisateurs puissent continuer à utiliser le logiciel, notamment dans un contexte pédagogique. La pérennité du logiciel passe aussi par l'amélioration et l'évolution du code. Aussi espérons nous que la bibliothèque *flip*, qui en constitue le noyau central, puisse un jour devenir *open-source* afin d'attirer de nouveaux contributeurs externes au projet pour nous aider dans cette tâche. La pérennité passe enfin par l'ajout de nouvelles fonctionnalités au logiciel qui permettront de soutenir d'autres types d'interaction et donc d'utilisation. Les retours obtenus de la part des premiers utilisateurs sont riches d'enseignements et nous permettent déjà d'envisager de nouvelles perspectives à donner au projet. Aussi souhaitons nous en obtenir de nouveaux, notamment à travers les prochaines utilisations prévues dans le cadre du MOOC à partir de novembre ou durant le cours donné sur le langage Faust avec Kiwi au second semestre universitaire prochain, afin d'alimenter une nouvelle fois cette boucle.

Parmi l'ensemble des perspectives de développement évoquées, celle du jeu collaboratif nous paraît être la plus importante à considérer pour la poursuite du projet. Aussi, l'un des enjeux principaux dans ce domaine se situe selon nous prioritairement dans l'amélioration du contrôle de paramètres dans une situation où plusieurs participants sont réunis au même endroit. Dans ce type précis de situation, la latence induite actuellement par l'objet *hub*, impliquant forcément une liaison au serveur central distant pour atteindre un autre participant, est peu soutenable. Il nous faudra donc penser de nouvelles solutions plus adaptées à cette configuration. L'une d'entre elle serait d'imaginer un objet permettant d'offrir une communication plus directe, par exemple en pair-à-pair, entre deux machines connectées à un même réseau local afin d'obtenir un temps de latence plus acceptable. Un mécanisme de découverte automatique de service pourrait aussi s'ajouter à ce système afin de faciliter l'interconnexion des utilisateurs présents sur le même réseau.

BIBLIOGRAPHIE

[Barbosa, 2003], Á. Barbosa, Displaced Soundscapes: A Survey of Network Systems for Music and Sonic Art Creation. *Leonardo Music Journal, 13*, pp. 53-59, 2003.

[Beaudouin-Lafon & Coutaz, 1995], M. Beaudouin-Lafon et J. Coutaz, Scoop / rapport de recherche 1994-1995. Technical report, SCOOP working group, Novembre 1995.

[Benson, 2012], A. Benson, *Collaboration in the Forum*, [Article de Blog], Cycling'74, publié 21 février 2012, [consulté le 1er juillet 2018]. Disponible à l'adresse : https://cycling74.com/articles/collaboration-in-the-forum.

[Chacon, 2009], S. Chacon, Pro Git. Books for professionals by professionals, Apress, p.231, 2009.

[Colafrancesco & al., 2013], J. Colafrancesco, P. Guillot, E. Paris, A. Sèdes et A. Bonardi, « La bibliothèque HOA, bilan et perspectives ». Actes des Journées d'Informatique Musicale (JIM), Saint-Denis, France, 2013.

[Dannenberg & al., 2007], R. B. Dannenberg, S. Cavaco, E. Ang, I. Avramovic, B. Aygun, J. Baek, E. Barndollar, D. Duterte, J. Grafton, R. Hunter, C. Jackson, U. Kurokawa, D. Makuck, T. Mierzejewski, M. Rivera, D. Torres, et A. Yu, « The Carnegie Mellon Laptop Orchestra » In *Proceedings of the 2007 International Computer Music Conference, Volume II*. San Francisco: The International Computer Music Association, pp. II-340-343, 2007.

[De Lavergne & Heïd, 2013], C. De Lavergne et M-C. Heïd, « Former à et par la collaboration numérique », *tic&société*, Formes et enjeux de la collaboration numérique, Vol. 7, n°1, 2013, [en ligne], [consulté le 18 septembre 2018]. Disponible à l'adresse : https://journals.openedition.org/ticetsociete/1308.

[Digiology, 2010a], Digiology, *Collaborative patching*, [Forum], Cycling'74, publié le 4 avril 2010, [consulté le 1er juillet 2018]. Disponible à l'adresse : https://cycling74.com/forums/collaborative-patching.

[Digiology, 2010b], Digiology, *MSP/Audio collaborative patch - intermediate*, [Forum], Cycling'74, publié le 5 avril 2010, [consulté le 1er juillet 2018]. Disponible à l'adresse : https://cycling74.com/forums/mspaudio-collaborative-patch-intermediate.

[Durham, 2012], M. Durham, *Collaborative patching.... ideas for sharing and group project development*, [Forum], Cycling'74, publié le 25 janvier 2012, [consulté le 1er juillet 2018]. Disponible à l'adresse : https://cycling74.com/forums/collaborative-patching-ideas-for-sharing-and-group-project-development/.

[Ellis & Gibbs, 1989], C. A. Ellis and S. J. Gibbs. « Concurrency Control in Groupware Systems », Proceedings of the ACM SIGMOD Conference on the Management of Data – SIGMOD'89, pp. 399-407, May 1989.

[Ellis & al., 1991], C. A. Ellis, S. J. Gibbs, G. Rein, Groupware - some issues and experiences. Commun. ACM 34, 1, New York, NY, USA, pp. 39-58, 1991.

[Ellis & Wainer, 1994], C. A. Ellis and J. Wainer, A conceptual model of groupware, In *ACM CSCW 94 conference on Computer Supported Cooperative Work*, pp. 79-88, 1994.

[Fielding, 2000], R. T. Fielding, « Architectural styles and the design of network-based software architectures », PhD Dissertation. Dept. of Information and Computer Science, University of California, Irvine, 2000. [disponible en ligne, consulté le 12/09/18] : https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[Galleron & al., 2018], P. Galleron, E. Maestri, J. Millot, A. Bonardi, E. Paris, « enseigner le *patching* de manière collective avec le logiciel collaboratif kiwi », Actes des Journées d'informatique Musicale (JIM), Amiens, France, pp. 106-114, 2018.

[Gellersen & Schmidt, 2002], H-W. Gellersen and A. Schmidt, Look who's visiting- supporting visitor awareness in the web, Int. J. Hum.-Comput. Stud., 56(1), pp. 25-46, 2002.

[Greenberg, 1990], S. Greenberg, « Sharing views and interactions with single-user applications », *In* Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Oce information systems, New York, NY, USA, pp. 227-237, 1990.

[Greenberg & al., 1996], S. Greenberg, C. Gutwin and M. Roseman, « Semantic Telepointers for Groupware », In Proceedings of the Sixth Australian Conference on Computer-Human Interaction - OZCHI'96, Hamilton, New Zealand, IEEE Computer Society Press, pp. 54-61, November 1996.

[Greenberg & Roseman, 1998], S. Greenberg and M. Roseman, *Groupware Toolkits for Synchronous Work,* In M. Beaudouin-Lafon (ed.), *Trends in CSCW*, John Wiley & Sons Ltd, 1998.

[Greenberg, 2002], S. Greenberg, « Real Time Distributed Collaboration », GroupLab Report, Dept. of Computer Science, University of Calgary, Canada, 2002.

[Graef, 2007], A. Graef, « Interfacing Pure Data with Faust », In LAC, editor, Linux Audio Conference, Berlin, Germany, 2007.

[Guillot, 2014], P. Guillot, « Une nouvelle approche des objets graphiques et interfaces utilisateurs dans Pure Data », Actes des Journées d'informatique Musicale (JIM), Bourges, France, 2014.

[Gutwin & Greenberg, 1997], C. Gutwin and S. Greenberg, *Workspace Awareness*, Position paper for the ACM CHI'97 Workshop on Awareness in Collaborative Systems, organized by Susan E. McDaniel and Tom Brinck, Atlanta, Georgia, 1997.

[Gutwin, 1997], C. Gutwin, « Workspace Awareness in Real-Time Distributed Groupware », PhD Dissertation, Department of Computer Science, University of Calgary, Canada, 1997.

[Haefeli, 2013], R. Haefeli, « netpd - a Collaborative Realtime Networked Music Making Environment written in Pure Data », In Proceedings of the 11th International Linux Audio Conference (LAC 2013), pp. 1-9, Graz, Autriche, 2013.

[Henri & Lundgren-Cayrol, 2001], F. Henry et K. Lundgren-Cayrol, « Apprentissage collaboratif à distance: pour comprendre et concevoir les environnements d'apprentissage virtuels », Presses de l'Université du Québec, Sainte-Foy, Québec, Canada, 2001.

[Hunt & McIlroy, 1976], J. W. Hunt et M. D. McIlroy, « An Algorithm for Differential File Comparison » *In. Computing Science Technical Report,* Bell Laboratories, Murray Hill, New Jersey, 1976.

[Jelmam, 2010], Y. Jelmam, « Travail collaboratif et interactions dans les forums de discussion fermés. Cas d'élèves ingénieurs tunisiens », *Questions vives*, 7 (14), recherches en éducation, pp. 89-105, 2010.

[Laliberté, 2013], M. Laliberté, « Émergence et développement de l'informatique musicale », In *Théories de la composition musicale au XX° siècle,* Symétrie, Lyon, 2013.

[Letz & al., 2013], S. Letz, D. Fober, Y. Orlarey, « Comment embarquer le compilateur Faust dans vos applications ? », Actes des Journées d'informatique Musicale (JIM), Paris, France, pp. 137-140, 2013.

[Myers, 1990], B. A. Myers, Taxonomies of visual programming and program visualization, In Journal of Visual Languages and Computing, Vol. 1, Iss. 1, pp. 97-123, 1990.

[McCaffrey, 1998], L. McCaffrey, Representing Change in Persistent Groupware Environments. Research report GroupLab, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, 1998.

[Orlarey & al., 2009], Y. Orlarey, D. Fober, and S. Letz, FAUST - an Efficient Functional Approach to DSP Programming, New Computational Paradigms for Computer Music, Editions DELATOUR FRANCE, 2009.

[Paris & al., 2017], E. Paris, J. Millot, P. Guillot, A. Bonardi, A. Sèdes, « Kiwi : vers un environnement de création musicale temps réel collaboratif, premiers livrables du projet Musicoll », Actes des Journées d'informatique Musicale (JIM), Paris, France, 2017.

[Pd~Graz, 2009], Collectif Pd~Graz, « blind date – an audiovisual performance by Pd~graz », Graz, Autriche, 2009, [consulté le 29 juillet 2018]. Disponible à l'adresse : http://pd-graz.mur.at/blind-date/blind_date.pdf.

[Peter & Trudy Johnson-Lenz, 1990], P. et T. Johnson-Lenz, *Rhythms, Boundaries, and Containers: Creative Dynamics of Asynchronous Group Life*, Awakening Technology Research Report n°4, April 1990. [consulté le 28 août 2018], Disponible à l'adresse : http://nexus.awakentech.com:8080/at/awaken1.nsf/UNIDs/CFB70C1957A686E98825654000699E1B?OpenDocument.

[Prakash & Knister, 1992], A. Prakash and M. J Knister, « Undoing actions in collaborative Work » In Proceedings of the ACM CSCW'92 Conference on Computer-Supported Cooperative Work, Toronto, Canada, pp. 273-280, 1992.

[Puckette, 1988], M. Puckette, « The Patcher », *In Proceedings of the International Computer Music Conference (ICMC)*, pp. 420-429, San Francisco, Californie, États-Unis, 1988.

[Puckette, 1996], M. Puckette « Pure Data: another integrated computer music environment », *In* proceedings of the International Computer Music Conference, 1996.

[Puckette, 2002], M. Puckette, « Max at Seventeen », Computer Music Journal 26:2, pp. 31-43., 2002.

[Puckette, 2009], M. Puckette, Étude de cas sur les logiciels pour artistes : Max/MSP et Pure Data, *In* David-Olivier Lartigaud, Ed., Art++, Editions Hyx, 2009.

[Roseman & Greenberg, 1992], M. Rosemann et S. Greenberg, « GroupKit: A groupware toolkit for building real-time conferencing applications », In Proceedings of the ACM CSCW Conference on Computer Supported Cooperative Work, ACM Press, Toronto, Canada, pp. 43-50, 1992.

[Roseman & Greenberg, 1996], M. Rosemann et S. Greenberg, « Building Real Time Groupware with GroupKit, A Groupware Toolkit », ACM Transactions on Computer Human Interaction - ACM TOCHI, ACM Press, New York, États-Unis, pp. 66-106, 1996.

[Salber, 1995], D. Salber, De l'interaction individuelle aux systèmes multiutilisateurs. L'exemple de la Communication Homme-Homme-Médiatisée, Thèse de doctorat Informatique, Université Joseph Fourier, Grenoble, France, pp. 303., 1995.

[Salber & al., 1995], D. Salber, J. Coutaz, D. Decouchant, M. Riveill, De l'observabilité et de l'honnêteté : Le cas du contrôle d'accès dans la Communication Homme-Homme Médiatisée, *In Proceedings of the* septième Journées sur l'Ingénierie et l'Interaction Homme-Machine, IHM'95, Toulouse, pp. 27-34, 1995.

[Sèdes & al., 2014], A. Sèdes, P. Guillot et E. Paris, « The HOA library, review and prospect », proceedings of the ICMC-SMC 2014, Grèce, Athènes, 2014.

[Sèdes & al., 2017], A. Sèdes, A. Bonardi, E. Paris, J. Millot, P. Guillot, « Teaching, investigating, creating: Musicoll », actes du colloque international Innovative Tools and Methods for Teaching Music and Signal Processing, sous la direction de L. Pottier, Presses des Mines, Paris, France, pp. 63-72, 2017.

[Sèdes & al., 2018], A. Sèdes, A. Bonardi, P. Guillot, E. Paris et J. Millot, « Créer, enseigner, chercher : MUSICOLL », *Revue Francophone d'Informatique Musicale* [En ligne], n°6 - Techniques et méthodes innovantes pour l'enseignement de la musique et du traitement de signal, Saint-Denis, France, Juin 2018. http://revues.mshparisnord.org/rfim/index.php?id=510

[Sun & al., 1998], C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, « Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems », ACM Transcription of Computer-Human Interaction, 5(1), pp. 63–108, 1998.

[Stefik & al., 1987], M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar, « WYSIWIS Revised: Early Experiences with Multi-User Interfaces », ACM Transactions on Office Information Systems 5(2), pp. 147-167, 1987.

[Tang, 1991], J. C. Tang, « Findings from Observational Studies of Collaborative Work », International Journal of Man Machine Studies, Academic Press Ltd. London, UK, 34(2): pp. 143-160, 1991.

[Touskalas, 2011], K. Touskalas, « A Collaboration Workflow from Sound-based Composition to Performance of Electroacoustic Music using “Pure Data” as a framework », Pure Data Convention, Weimar-Berlin, Allemagne, 2011.

[Walckiers & De Praetere, 2004], M. Walckiers, et T. De Praetere, L’apprentissage collaboratif en ligne, huit avantages qui en font un must. *Distances et savoirs*, *2* (1), pp. 53-75, 2004.

[XH9O, 2011], XH9O, *Hippie Patching*, [Forum], Cycling’74, publié le 28 juillet 2011, [consulté le 1er juillet 2018]. Disponible à l’adresse : https://cycling74.com/forums/hippie-patching.

[Zicarelli, 2002], D. Zicarelli, « How I Learned to Love a Program That Does Nothing », *Computer Music Journal*, vol. 26, no. 4, pp. 44-51, 2002.

[Zmölnig, 2007], IO. M. Zmölnig, « patching music together: collaborative live coding in pure data », In proceedings of the *Pd Convention 2007*, Montréal, Québec, Canada, 2007.