# FONDATIONS POUR UN *MIDDLEWARE* TEMPOREL ADAPTÉ AU SPECTACLE VIVANT

*Martin Fouilleul*
STMS – Sorbonne Université, IRCAM, CNRS
martin.fouilleul@ircam.fr

**RÉSUMÉ**

Les arts vivants sont par nature pris dans un flux temporel, et font un usage créatif du temps. Par conséquent, les interactions temporelles jouent un rôle essentiel dans une représentation, autant du point de vue des artistes, que de celui des techniciens ou du public. Bien que ces interactions relèvent principalement de la créativité et du savoir-faire humain, certains scénarios peuvent bénéficier d'un apport logiciel. Nous exposons ici le besoin d'un écosystème logiciel permettant de spécifier et réaliser des interactions temporelles humain-machine et machine-machine, et nous suggérons une approche fondée sur un *middleware* permettant de connecter des outils existants dans un contexte temps-réel. Nous identifions certains composants clés du problème, à savoir la découverte de service, l'élection et la synchronisation d'horloge, et le passage de messages structurés entre services. Nous décrivons ensuite l'implémentation des prototypes que nous avons écrits pour chacun de ces composants. Ces prototypes donnent des résultats encourageants. Ceux-ci prennent la forme d'une bibliothèque écrite en C, et exposée sous forme de *plug-ins* ou de *bindings* dans d'autres environnements, comme *Max/MSP* ou Python.

## 1. INTRODUCTION

Le temps est un élément crucial des différentes formes du spectacle vivant. Que ce soit la musique, l'opéra, le théâtre, la danse, ou encore les installations interactives, toutes ces formes sont par nature prises dans un flux temporel, et font un usage créatif du temps pour mettre en forme leur dramaturgie, dérouler leur action, créer des climax et des contrastes.

L'art de l'interprète consiste en grande partie à jouer avec le temps. Certains évènements (gestes, sons, etc.) doivent se produire à un moment précis, dans un certain ordre, à une vitesse donnée, pour créer l'effet désiré. Ces évènements doivent également maintenir entre eux des relations temporelles complexes (on peut par exemple penser aux doigts et à l'archet d'un violoniste, où aux répliques et aux gestes d'un comédien). La perception du temps est également de la plus haute importance, permettant l'anticipation, la surprise, la tension et la résolution.

Les musiciens, danseurs, comédiens, doivent interpréter un flux temporel abstrait décrit sous la forme d'une partition ou d'un scénario, ou bien construire la progression temporelle sur le vif, dans le cas des formes ouvertes ou de l'improvisation. Mais dès lors qu'ils jouent à plusieurs, ils doivent aussi s'accorder, jusqu'à un certain point, sur une notion commune du temps : ils doivent se synchroniser. Il est à noter qu'ils doivent le faire en *live*, tout en retenant leur capacité expressive. Par exemple, certaines parties du flux temporel peuvent diverger et s'écouler indépendamment, pour se rejoindre sur certains points de synchronisation dont la date absolue n'est pas décidable *a priori*. C'est pourquoi les flux temporels de spectacles se limitent rarement à des lignes temporelles (comme la *timeline* d'un séquenceur).

Évidemment, les dispositifs techniques du spectacle sont partie prenante de ce réseau de relations temporelles complexes et doivent également se synchroniser et réagir à la progression de l'action scénique. Ils peuvent même être directement intégrés à l'œuvre sous une forme interactive et devenir un protagoniste de la pièce. La rencontre entre l'artiste humain et la machine, dans ce contexte, dépasse le cadre de l'utilitarisme ou de la sujétion pour offrir la potentialité d'une réelle collaboration.

Prenons l'exemple d'un opéra comme *Donnerstag aus Licht* de K. Stockhausen, qui s'inscrit dans un projet esthétique de *Gesamtkunstwerk* (œuvre d'art total). Cette pièce requiert la synchronisation de plusieurs média temporels, incluant deux orchestres, des chanteurs d'opéra, un chœur, des danseurs, ainsi qu'une projection vidéo, un dispositif de sonorisation et une partie musicale électronique, des jeux de lumière, des décors, des sous-titres, etc. Il existe une multitude de relations de dépendances temporelles entre ces différents postes, qui se réalisent à travers des mécanismes d'interaction eux-mêmes variés : indices visuels ou acoustiques, communication verbale à travers un système d'intercoms, ou protocoles informatiques tels que MIDI ou OSC.

Loin d'être des tâches uniquement techniques, tous ces arrangements pratiques débouchent sur des questions fondamentales en informatique musicale. Celles-ci incluent la synchronisation chronométrique et l'horodatage de flux de données, la spécification de contraintes temporelles, la détection d'évènements distribués, et la réalisation de ces tâches dans un contexte temps réel avec des ressources limitées.

Pourtant, ces problèmes sont encore habituellement traités de manière *ad hoc*, en s'appuyant sur le savoir-

faire et l'interaction directe des opérateurs. Ainsi, la plupart des situations réelles ne bénéficient pas autant qu'elles le pourraient des avancées de la recherche. Or, bien que nous pensons que le savoir-faire et l'interaction humaine resteront un composant clé de la création artistique dans le futur, nous devons reconnaître que certaines situations sont difficilement réalisables en s'appuyant uniquement sur les compétences humaines, ce qui peut parfois limiter la liberté créatrice de l'artiste. C'est le cas en particulier des situations nécessitant une réponse temps réel, ou bien le suivi simultané de plusieurs flux d'évènements complexes. Le besoin de solutions technologiques est d'autant plus prononcé lorsqu'il s'agit de synchroniser un ensemble mêlant machines et interprètes humains en interaction. Cette thématique est présente dans d'autres champs, où elle est parfois désignée sous le nom de *cyber-physical interaction with human in the loop* [34]. Dans le champ musical, elle s'exprime par exemple dans la musique mixte, dans laquelle des instrumentistes humains jouent *avec* (et non plus seulement *sur*) une partie électronique.

Un nombre croissant d'artistes sont intéressés par ce type d'interaction humains–machines, qui dit en effet quelque chose d'une société de plus en plus intégrée avec sa propre technologie. A titre d'exemple, on pourra se référer à *Études pour théâtre acoustique* pour instrument automate de Pedro Garcia Velasquez, à l'installation interactive *Biotope* de Jean-Luc Hervé, ou encore aux œuvres de musique mixte de Sasha Blondeau comme *Urphänomen*. En conséquence, il y a un réel besoin à la fois d'outils et de concepts, qui permettront aux artistes, aux interprètes et aux techniciens de demain de concevoir et réaliser des œuvres d'art empuissantisées [1] par la technique, et par là même capable de porter sur elle un regard critique.

## 2. VERS UN *MIDDLEWARE* POUR LE SPECTACLE VIVANT

Les situations artistiques évoquées plus haut mettent en lumière le besoin d'un écosystème d'outils permettant d'interconnecter et de synchroniser les différents corps de métier d'un spectacle vivant. Elles nous poussent à explorer une approche distribuée, extensible et modulaire, de type *middleware*. Cette approche a déjà été explorée dans d'autres champs, comme la robotique, avec des systèmes comme ROS [38] ou Yarp [26], ou pour les applications distribuées avec DDS [36]. Il s'agit donc ici de déterminer les composants essentiels permettant à terme de construire un *middleware* adapté au spectacle vivant.

Il existe déjà des logiciels de régie utilisés pour contrôler et coordonner les éléments d'un spectacle. Leur approche peut être fondée sur la notion de *timelines*, comme *Medialon* [25], sur la notion de *cuelists*, comme *QLab* [9], ou encore présenter un modèle hybride comme *Ossia Score* [6]. Ce dernier a également la particularité d'être construit sur un modèle objet distribué inspiré du protocole OSC, et exposé à travers la bibliothèque libossia [19]. Cela lui permet de communiquer à travers plusieurs protocoles dérivés d'OSC (comme OSCQuery ou Minuit) avec des appareils ou logiciels distants. Pour notre part nous nous intéressons ici à l'interaction temporelle entre des logiciels dans une approche distribuée, et nous nous concentrerons donc sur *l'infrastructure* permettant ces interactions plutôt que sur les modèles de conduite de spectacle proposés par ces logiciels.

L'idée d'un *framework* destiné à faciliter la construction d'expériences audiovisuelles interactives a également déjà été développée par le projet Soundworks [22], qui propose une architecture client–serveur utilisant l'environnement javascript Node et controlée *via* des interfaces web. Nous explorons ici une architecture plus décentralisée et indépendante des technologies web.

### 2.1. Propriétés attendues

Nous pensons que pour être adoptées par les praticiens, les solutions proposées doivent s'intégrer aux *workflows* existants. Elles doivent être suffisamment flexibles pour permettre tous les arrangements *ad hoc* et les idiosyncrasies qui font de chaque spectacle une œuvre unique. Elles doivent donc être *modulaires* et *extensibles*.

La nature *distribuée* d'un tel système est un corollaire de la division des tâches entre les différents corps de métiers du spectacle. D'autre raisons rendent également désirable une solution distribuée. L'une d'elle est évidemment la puissance de calcul : les logiciels multimédia peuvent être gourmands en ressources et le besoin de stabilité et de réponses temps-réel poussent à allouer des machines dédiées à ces différents outils. Il est également souhaitable d'éviter de créer des nœuds de congestion ou des points de défaillance centralisés.

Il convient de noter que la distribution des outils sur plusieurs machines doit être transparente, et que la localisation spécifique d'un outil ne devrait pas être déterminante du point de vue des autres outils. En d'autre termes, le système devrait être *agnostique à l'infrastructure*. C'est une propriété importante dans un contexte artistique, où les contraintes et les exigences peuvent varier de manière significative au cours du processus de création.

Les solutions proposées doivent permettre aux différents outils de communiquer et d'interagir au travers de mécanismes variés. On peut par exemple envisager des canaux de publication-souscription permettant d'échanger et de multiplexer de l'information, des appels de procédures distantes, des données partagées, ou encore des *locks* et des points de rendez-vous permettant de synchroniser plusieurs processus.

[1] Nous empruntons ici un néologisme introduit par Frédéric Lordon dans *Les affects de la politique*, faisant écho à la notion d'*empowerment*, qui désigne la capacité d'un groupe ou d'individus à s'autonomiser et à exercer leur pouvoir en prenant le contrôle de leurs conditions d'existence. Nous pensons que la référence métaphorique à ces thématiques est fertile pour penser les possibilités critiques d'un art parfois assujetti, ou du moins démuni, face à la pénétration des artefacts techniques dans les rapports sociaux.

### 2.2. Format d'échange de données

Un format d'échange de données est essentiel pour communiquer des informations entre les applications utilisant le *middleware*, d'autant plus si elles sont écrites de manière indépendante, dans des langages différents, ou pour des plateformes différentes. Plusieurs approches sont possibles :

— L'utilisation d'un format texte directement lisible, tel que XML [5] ou JSON [4], présente des avantages en terme d'expérience utilisateur, au détriment de la performance et de la complexité des modules de désérialisation.

— Les langages de définition d'interface tels que OMG IDL [37] ou Google Protocol Buffers [13] permettent des implémentations plus efficaces et fortement typées, ainsi que la génération automatique de code de sérialisation/désérialisation. Ils nécessitent cependant un accord préalable entre les modules désirant échanger des informations, sous la forme de fichiers description d'interfaces. Ils se prêtent donc bien au développement d'applications distribuées bien spécifiées, mais sont plus contraignants dans le cadre d'un processus créatif pouvant faire varier fréquemment ces spécifications.

— Des protocoles spécifiques aux applications musicales existent, comme MIDI [27] ou OSC [39]. Le protocole MIDI nous semble trop spécifique au contrôle d'instruments musicaux pour notre cas d'usage. En revanche, le protocole OSC offre une solution simple et extensible qui a le mérite d'être très répandue dans les applications audio et les contrôleurs musicaux.

Le protocole OSC semble donc un choix avantageux pour gérer les échanges de données entre les modules de notre *middleware*.

Cependant, OSC souffre de certaines limites quand au transport de données fortement structurées. En effet, sa seule manière de construire une hiérarchie est basée sur la notion de *bundle*, qui est une collection non structurée de messages. Celle-ci n'autorise pas l'expression de structures imbriquées de type clé–valeur, un *bundle* n'étant lui même associé à aucune clé. Cette restriction témoigne selon nous du fait qu'OSC a été initialement conçu pour le contrôle d'applications musicales, sous la forme d'envoi d'ordres ou de paramètres : ainsi sa structure correspond directement à un appel de fonction, mais est beaucoup moins adaptée à la description générale de données.

Afin de pallier cette limite, nous proposons d'étendre le protocole par l'ajout d'un tag « `o` » permettant d'insérer un message comme argument d'un autre message, et d'abandonner la notion de *bundle*.

Le protocole OSC souffre également d'autre limites, comme la sous-spécification du type `timetag` [10], la difficulté d'ajouter de nouveaux types natifs, ou l'absence de mécanismes visant à rendre possible la compatibilité ascendante. A ce stade, nous ne sommes pas entièrement convaincu de la pertinence à long terme du protocole OSC en tant que format général d'échange de données au sein d'un *middleware temporel*. Cependant, son intégration dans une large gamme de produits existants restera un argument fort tant qu'une alternative clairement avantageuse n'aura pas été proposée.

### 2.3. Découverte de services

Nous avons affirmé plus haut que les solutions proposées devraient être agnostiques vis-à-vis de l'infrastructure. Cela signifie que dans la mesure du possible, les connections entre services ne devraient pas nécessiter de reconfiguration lors d'un changement d'infrastructure. En particulier, un processus devrait pouvoir utiliser les services offerts par d'autres processus sans que l'opérateur ait à spécifier la localisation de ces services, tel que leurs adresses IP ou leur ports. Un service devrait donc être identifié par un nom et un type, indépendamment du processus particulier qui offre ce service et de la machine particulière sur laquelle il s'exécute.

Plusieurs protocoles ont été proposés pour la découverte de services, comme SSDP [1], WSDD [35], ou NetBios [33]. Une autre stratégie répandue est d'utiliser des enregistrements mDNS d'un serveur de noms de domaines [32]. La découverte de service fondée sur DNS [7] est par exemple implémentée par le protocole Bonjour [2] dans les produits Apple.

Bien que l'adoption d'un protocole existant comme Bonjour semble une solution attrayante, elle nécessite une infrastructure dont nous ne pouvons pas présupposer la présence sur tous les systèmes visés. Même si cet argument est de moins en moins fort au vu de la propagation de Bonjour au-delà de l'écosystème Apple, il nous semble important de proposer une solution autonome, notamment pour les systèmes embarqués. La dépendance à un serveur DNS introduit également un point de défaillance centralisé que nous souhaitons éviter. Enfin, l'utilisation de mDNS impose l'unicité des noms de service, or nous verrons que le relâchement de cette contrainte nous permet d'implémenter très facilement des mécanismes de passage de message de type publication/souscription. Nous avons donc opté pour un système de découverte de service *ad hoc* qui sera décrit en sous-section 3.2.

### 2.4. Horodatage de flux

Comme l'objectif du *middleware* est d'échanger des informations à caractère fortement temporel, il est particulièrement important que le système maintiennent une notion cohérente du temps à travers l'ensemble des processus participants. La distribution d'une horloge commune n'est pas un problème trivial, et les approches simples comme l'algorithme de Berkeley [14] ou l'algorithme de Cristian [8] ne fournissent pas une précision suffisante.

Dans le champ de l'informatique musicale, ce problème est souvent esquivé au prétexte que l'oreille humaine ne peut généralement distinguer des déplacements rythmiques ou des latences inférieures à 10 ms [11, 18]. Nous considérons cet argument comme faible pour plusieurs raisons :

— Il ignore le fait que des décalages de phase de moins de 10 ms, bien qu'elle ne soient pas perçues comme des distorsions rythmiques, produisent des différences audibles tel que des filtrages en peigne, ou des artefacts de spatialisation comme l'effet Haas [15].
— De faibles erreurs temporelles instantanées, si une grande attention n'est pas portée à les compenser, peuvent s'accumuler et conduire à des différences perceptibles, particulièrement durant le temps long d'un spectacle.
— Les approches simples sont exposées à des pics de latence et de gigue d'horloge, qui peuvent mener à des erreurs significatives.
— L'argument psycho-acoustique n'est pas pertinent pour des messages de contrôle que nous aurions besoin d'horodater, par exemple pour en reconstituer la séquence lorsque la transmission des messages ne conserve pas nécessairement leur ordre.

Nous pensons que la capacité de maintenir une horloge distribuée avec une précision plus importante est fondamentale pour un *middleware temporel*, et que le problème n'a pas été suffisamment exploré dans notre domaine. En revanche, il a été étudié systématiquement dans d'autres champs, et nous pouvons bénéficier de cette expérience.

Le standard le plus répandu pour la synchronisation d'horloges à travers un réseau à paquets commutés est NTP [28]. Il a été continuellement affiné depuis sa conception jusqu'à la quatrième et actuelle version [31], et s'est montré remarquablement robuste et précis pour un grand nombre d'applications. Il permet de distribuer une horloge avec une précision de quelques millisecondes à travers le réseau Internet, et atteint habituellement une précision inférieure à la milliseconde sur un réseau local [29].

Un réseau NTP est une hiérarchie auto-organisée de serveurs et de clients d'horloge. Chaque noeud peut se synchroniser en tant que client sur plusieurs serveurs, et redistribuer l'horloge à d'autres clients plus bas dans la hiérarchie. Un client utilise plusieurs techniques de mitigation pour compenser les erreurs dues à la latence, à la gigue, et l'asymétrie du réseau.

Le protocole SNTP [30] est une version simplifiée de NTP destinée à des serveurs ayant une horloge de référence unique, ou à des clients ayant un seul serveur de référence et qui ne redistribuent pas leur horloge à d'autre clients. Il calcule directement une estimation du décalage entre le client et le serveur grâce aux messages échangés entre ceux-ci et n'utilise aucune forme de mitigation.

Le protocole PTP [17] est destiné aux mesures de très haute précision dans les systèmes de contrôle, et peut atteindre une précision inférieure à la micro-seconde. Il utilise pour cela une combinaison de logiciel et de matériel réseau dédiée, permettant d'estimer et de compenser la latence et la gigue du réseau, en modifiant l'horodatage des messages durant leur propagation sur le réseau.

Il nous parait bénéfique de s'appuyer sur la théorie et le savoir pratique renfermés dans un protocole éprouvé comme NTP. Cependant, comme dans le cas de la découverte de service, déléguer la synchronisation d'horloge à une implémentation existante de NTP nous paraît peu pratique, car celle-ci dépendrait de l'existence d'une infrastructure préalablement installée et correctement configurée sur toutes les machines. De plus, nous pouvons tirer parti de notre système de découverte de service et d'élection pour rendre le système de synchronisation d'horloge robuste à la défaillance de l'horloge maître, sans avoir à configurer des serveurs de secours. Enfin, nous souhaitons à terme adapter les paramètres et les constantes de temps du protocole, initialement prévues pour le réseau Internet, à notre cas d'usage, de manière à atteindre des convergences plus rapides ou des précisions plus grandes (éventuellement au détriment de la bande passante).

## 3. IMPLÉMENTATION

### 3.1. Bibliothèque OSC

Comme mentionné en sous-section 2.2, nous utilisons une version légèrement étendue du protocole OSC pour échanger des données entre les composants de notre *middleware*. Plusieurs bibliothèques OSC sont disponibles pour manipuler des messages OSC, comme `libo` [23], `liblo` [16], `oscpack` [3] ou `rtosc` [24]. Une grande diversité existe dans leurs approches, leurs capacités, et leurs performances, mais la plupart amalgame les concepts liés au format de données (composer et analyser des messages), le protocole de transport (envoyer et recevoir des paquets), et la distribution de ces messages vers les fonctions de traitement appropriées.

Nous avons opté pour l'écriture d'un bibliothèque OSC autonome pour plusieurs raisons :

— L'extension que nous proposons s'intégre mal avec les choix d'API et d'implémentation des bibliothèques existantes.
— Notre utilisation d'OSC se résume à celui d'un format d'échange de données et les bibliothèques mentionnées rendent peu pratique l'utilisation de cette fonctionnalité sans utiliser aussi leurs interfaces de transport et d'exécution.
— Du fait de leur choix d'API et d'implémentation visant une plus grande généralité, ces bibliothèques exhibent souvent de faibles performances.

#### *3.1.1. Composition*

Notre implémentation permet de composer des messages et des *bundles* OSC en place, dans des tableaux fournis par l'utilisateur. Les structures exposées par l'API ne contiennent que des pointeurs désignant des emplacements à l'intérieur de ces tableaux. De cette manière nous évitons entièrement les allocations mémoire à l'intérieur de la bibliothèque, ce qui s'est avéré crucial pour la performance.

Une première API permet de composer pas-à-pas des éléments OSC, tandis qu'une deuxième du type `printf()` peut être utilisée pour créer des messages en une seule

passe lorsque la chaîne de type et tous les arguments sont disponibles immédiatement.

#### *3.1.2. Analyse*

La désérialisation de messages OSC s'effectue avec les même structures que la composition et s'effectue également à partir de tableaux fournis par le code client. La bibliothèque expose une API de type *itérateur* qui permet de parcourir la structure du paquet OSC et d'en extraire les informations utiles, et une API de type `scanf()`, qui permet de vérifier et d'extraire en une seule passe les arguments d'un message pour une chaîne de type donnée.

#### *3.1.3. Performances*

Nous avons réalisé plusieurs comparaisons de performance entre notre implémentation et celles des bibliothèques OSC mentionnées plus haut. Nous avons mesuré les tâches suivantes :

— composer un message OSC ;
— composer un *bundle* OSC contenant deux messages ;
— analyser un message OSC et extraire son adresse, sa chaîne de type et ses arguments ;
— analyser un *bundle* OSC contenant deux messages, et extraire les adresses, les chaînes de type et les arguments.

Le message utilisé consiste en l'adresse OSC `/foo/bar`, l'argument `"Hello, world !"`, un argument de type entier sur 64 bits, et un argument de type flottant en double précision. Chaque bibliothèque a été compilée depuis les sources en utilisant le même niveau d'optimisation (`-O3`), et liée de manière statique au programme de test. Les tâches ont été répétées cent millions de fois dans une boucle et le temps total d'exécution consigné dans un fichier. Le tableau 1 montre les résultats des tests. Pour chaque tâche, nous donnons le temps total, le nombre d'opérations par secondes, la durée moyenne d'une opération, et le facteur d'accélération par rapport à la plus lente implémentation. Nous mentionnons également le facteur d'accélération de notre implémentation par rapport à la plus rapide des alternatives (en l'occurence `oscpack`). `liblo` n'offre pas de moyen d'analyser directement des bundles, et n'apparaît donc pas dans cette tâche particulière.

### 3.2. Découverte de service

Un processus ayant besoin d'interagir avec d'autre processus présents sur le réseau peut utiliser le module de découverte de service intégré dans notre bibliothèque. Un service est connu à travers le réseau sous la forme d'un descripteur de service, qui comprend un nom, un type, un identifiant globalement unique, et une adresse à laquelle contacter ce service. Un processus peut également publier auprès de son module de découverte un descripteur de service, qui sera propagé à tous les autres modules. Un processus peut également demander une liste des services présents sur le réseau au module de découverte de services, et filtrer cette liste par nom ou par type. Enfin, un processus peut demander à être notifié quand un service apparaît ou disparaît des tables de descripteurs de son module de découverte.

#### *3.2.1. Découverte, Publication et révocation*

Lorsqu'un module de découverte est lancé, il rejoint un groupe *multicast* pour envoyer et recevoir des requêtes, et ouvre un port UDP éphémère pour recevoir les réponses des autres modules. Il s'exécute ensuite dans un *thread* séparé et envoie des requêtes de découverte sur l'adresse *multicast*, sous la forme d'un message OSC `/hello`, dont l'argument est son port de réponse.

Un module qui reçoit un message `/hello` renvoie une série de réponses pour chaque service enregistré localement. Ces réponses prennent la forme d'un message OSC `/publish` dont les arguments constituent un descripteur de service.

Lorsqu'un processus publie un descripteur de service auprès de son module de découverte de service, ce dernier envoie une série de messages OSC `/publish` sur l'adresse multicast.

Lorsqu'un processus arrête de fournir un service précédemment publié, il peut révoquer son descripteur de service auprès du module de découverte. Ce dernier envoie alors une série de messages `/revoke` associés à l'identifiant unique du service.

#### *3.2.2. Réduction de la congestion*

Pour éviter des pics d'encombrement du réseau, les messages de publication et de révocations sont envoyées un nombre limité de fois, à des intervalles de temps suivant une série géométrique, avec un délai initial aléatoire. Les messages de découverte sont d'abord envoyés à des intervalles de temps croissant géométriquement, puis de manière périodique passé un certain intervalle.

#### *3.2.3. Expiration*

Dans le cas où un processus s'arrêterait abruptement, il est possible que les services qu'il a précédemment publiés ne soient pas révoqués. Un processus qui effectue une recherche de services auprès de son module de découverte doit donc prévoir le cas où certains résultats représentent des services inactifs. Il est cependant souhaitable de limiter le nombre de services inactifs présents dans les tables de descripteurs de services de chaque module, ceci afin d'éviter des temps de recherche trop longs. C'est pourquoi le module de découverte de service associe à chaque descripteur une date d'expiration, qui est repoussée chaque fois qu'il reçoit un message `/publish` pour ce descripteur. Lorsque la date d'expiration est atteinte, le descripteur est silencieusement retiré de la table du module de découverte.

| Implementation | time (s) | Op length ($\mu$s) | Throughput (op/s) | Speedup vs. slowest | Speedup vs. oscpack |
|---|---|---|---|---|---|
| Composing $1 \times 10^8$ messages | | | | | |
| Blitz push | 4.38 | 0.044 | 22823059 | 30.11 | 1.52 |
| Blitz format | 4.14 | 0.041 | 24154397 | 31.86 | 1.61 |
| oscpack | 6.68 | 0.067 | 14980604 | 19.76 | 1.00 |
| rtosc | 7.97 | 0.080 | 12549376 | 16.55 | 0.84 |
| liblo | 75.68 | 0.757 | 1321303 | 1.74 | 0.09 |
| libo | 131.91 | 1.319 | 758065 | 1.00 | 0.07 |
| Composing $1 \times 10^8$ bundles | | | | | |
| Blitz push | 10.26 | 0.103 | 9745841 | 24.92 | 1.40 |
| Blitz format | 9.89 | 0.099 | 10107700 | 25.85 | 1.45 |
| oscpack | 14.36 | 0.144 | 6964290 | 17.81 | 1.00 |
| rtosc | 34.58 | 0.346 | 2892121 | 7.40 | 0.42 |
| liblo | 235.76 | 2.358 | 424164 | 1.08 | 0.06 |
| libo | 255.74 | 2.557 | 391028 | 1.00 | 0.06 |
| Parsing $1 \times 10^8$ messages | | | | | |
| Blitz iterator | 2.94 | 0.029 | 34013826 | 20.49 | 1.44 |
| Blitz scan | 3.14 | 0.031 | 31875865 | 19.20 | 1.35 |
| oscpack | 4.22 | 0.042 | 23689203 | 14.27 | 1.00 |
| rtosc | 13.18 | 0.132 | 7584987 | 4.57 | 0.32 |
| liblo | 60.24 | 0.602 | 1660161 | 1.00 | 0.07 |
| libo | 30.69 | 0.307 | 3258812 | 1.96 | 0.14 |
| Parsing $1 \times 10^8$ bundles | | | | | |
| Blitz iterator | 7.74 | 0.077 | 12918467 | 6.44 | 1.35 |
| Blitz scan | 8.52 | 0.085 | 11732967 | 5.85 | 1.23 |
| oscpack | 10.44 | 0.104 | 9574139 | 4.77 | 1.00 |
| rtosc | 34.76 | 0.348 | 2876787 | 1.43 | 0.30 |
| libo | 49.85 | 0.499 | 2005921 | 1.00 | 0.21 |

**Table 1.** Comparaisons de la vitesse d'exécution de plusieurs bibliothèques OSC.

### 3.3. Protocole d'élection

Un grand nombre d'applications distribuées nécessitent de désigner certains processus pour coordonner l'action des autres processus. C'est le cas en particulier des protocoles de synchronisation d'horloge, dans lesquels un ou des serveurs de temps permettent de synchroniser des processus clients.

La désignation manuelle d'un serveur maître impose à l'utilisateur une tâche de configuration supplémentaire. De plus, son aspect statique est peu adaptée aux situations ou la composition du réseau peut changer fréquemment, en particulier si le maître peut disparaître du réseau, car il faut alors lancer un nouveau serveur maître et reconfigurer tous les clients. Pour la même raison elle est peu robuste en cas de défaillance du maître. C'est pourquoi nous décrivons dans ce qui suit un protocole d'élection de maître, permettant à des processus de sélectionner automatiquement un processus maître. Nous utilisons en particulier ce protocole pour désigner une horloge maître pour notre système de synchronisation d'horloges.

Notre protocole d'élection doit prendre en compte non seulement la possibilité de messages perdus ou retardés, mais aussi celle de processus faillibles pouvant joindre ou quitter l'élection abruptement. C'est pourquoi les garanties de réception des messages offertes par le protocole TCP ne sont pas suffisantes pour rendre le problème trivial. Le protocole doit être construit *ab initio* autour de la possibilité de défaillances d'un des participants à l'élection.

Nous nous appuyons pour cela sur [20], sans toutefois distribuer à chaque participant une table d'activité décrivant le statut actif ou inactif de ses pairs. En effet, dans notre cas les différents processus sont considérés comme indépendants les uns des autres, et n'ont pas à connaître le statut de leurs pairs, à l'exception du processus qui sera élu comme maître. Une autre différence est que nous nous appuyons sur le module de découverte de service décrit précédemment pour constituer une liste des participants à chaque nouvelle élection, plutôt que de supposer une connaissance préalable de l'ensemble des participants potentiels.

Un participant au processus d'élection peut se trouver dans l'un des états suivants : `NORMAL`, `COHORT`, `LEADER`, `CANDIDATE`, `CLAIM`, ou `CONFIRM`. Lorsqu'une élection a été menée à son terme et que le processus est stabilisé, un seul participant se trouve dans l'état `LEADER` et les autres sont dans l'état `NORMAL`.

Chaque participant $P_i$ est associé à une adresse $A_i$ et un score unique $S_i$, et possède une variable $B$ stockant le couple $(A_b, S_b)$ du meilleur candidat du point de vue de ce participant. Un candidat possède par ailleurs une liste de pairs, qui indique pour chaque participant potentiel à l'élection son adresse, son statut actif ou inactif, et si ce participant à accepté sa candidature. Les messages que s'envoient les participant à l'élection prennent la forme de messages OSC dont les arguments forment un couple $(A_j, S_j)$. Pour chaque état, nous détaillons ci-dessous les évènements ou les messages auxquels peut réagir un participant $P_i$.

#### 3.3.1. *NORMAL ou* `LEADER`

— Démarrage, ou défaillance du maître : le participant demande au module de découverte de service une liste de pairs potentiels. Il envoie à chacun de ses pairs les message `/candidate`$(A_i, S_i)$, déclenche un timer, puis passe dans l'état `CANDIDATE`.

— `/candidate`$(A_j, S_j)$ : si $S_j < S_i$, $P_i$ déclenche une élection en se portant candidat, comme décrit plus haut. Si $S_j > S_i$, le participant stocke $(A_j, S_j)$ dans $B$, répond par un message `/ack`$(A_i, S_i)$, déclenche un timer, et passe dans l'état `COHORT`.

#### 3.3.2. *CANDIDATE*

— `/candidate`$(A_j, S_j)$ : si $S_j < S_i$, le candidat répond par un message `/reject`$(A_i, S_i)$. Si $S_j > S_i$, le partipant stocke $(A_j, S_j)$ dans $B$, répond par un message `/ack`$(A_i, S_i)$, déclenche un timer, et passe dans l'état `COHORT`.

— `/reject`$(A_j, S_j)$ : le candidat stocke $(A_j, S_j)$ dans $B$, répond par un message `/ack`$(A_i, S_i)$, déclenche un timer et passe dans l'état `COHORT`.

— `/ack`$(A_j, S_j)$ : le candidat marque $P_j$ comme actif dans sa liste de pairs.

— timeout : le candidat envoie un message `/claim`$(A_i, S_i)$ à tous les pairs marqués comme actifs, puis déclenche un timer et passe dans l'état `CLAIM`.

#### 3.3.3. *CLAIM*

— `/accept`$(A_j, S_j)$ : le candidat marque $P_j$ comme acceptant sa candidature. Si tous les pairs actifs ont accepté la candidature, le candidat envoie à ces pairs un message `/finish`$(A_i, S_i)$ puis passe dans l'état `LEADER`.

— timeout : Si tous les pairs actifs n'ont pas accepté la candidature à l'issue du timeout, l'élection doit être relancée.

#### 3.3.4. *COHORT*

— `/candidate`$(A_j, S_j)$ : si $S_j < S_b$, le participant répond par un message `/reject`$(A_b, S_b)$. Si $A_j > S_b$, le participant stocke $(A_j, S_j)$ dans $B$, réinitialise son timer, et répond par un message `/ack`$(A_i, S_i)$.

— `/claim`$(A_j, S_j)$ : si $j = b$, le participant répond par un message `/accept`$(A_i, S_i)$, déclenche un timer et passe dans l'état `CONFIRM`. Si $j \neq b$, le message est ignoré.

— timeout : si le participant n'a pas reçu de message `/claim`$(A_b, S_b)$ à l'expiration de son timer, le meilleur candidat est considéré comme défaillant et l'élection est relancée.

#### 3.3.5. *CONFIRM*

— `/finish`$(A_j, S_j)$ : si $j = b$, le participant passe dans l'état `NORMAL`. Sinon, l'élection est relancée.

— timeout : si le participant n'a pas reçu de message `/finish`$(A_b, S_b)$ à l'expiration de son timer, le meilleur candidat est considéré comme défaillant et l'élection est relancée.

### 3.4. Synchronisation

Comme mentionné précédemment, notre protocole de synchronisation d'horloge s'appuie sur NTP, avec toutefois quelques différences notables.

Notre protocole utilise les algorithmes de mitigation d'erreurs dues à la latence et à la gigue décrit dans la specification de NTP. En revanche, il ignore les aspects liés à la sécurité (chiffrement des messages et authentification des serveurs de temps), ainsi que l'organisation hiérarchique en strates de serveurs et de clients. Notre réseau de synchronisation se présente donc sous la forme de clients qui se synchronisent indépendamment sur un maître commun.

Nous exploitons les modules de découverte de services et d'élection de maître décrit en sous-section 3.3 pour désigner une horloge maître unique. Nous éliminons ainsi la tâche de configuration manuelle d'un serveur de temps. Cela permet également d'être robuste à la disparition (accidentelle ou volontaire) de l'horloge maître, en élisant une nouvelle horloge lorsqu'une absence du serveur maître est détectée.

Notre protocole maintient pour chaque processus client une *horloge utilisateur*, dérivée de l'horloge de l'hôte sur lequel le processus s'exécute. Le protocole ajuste uniquement les paramètres de cette horloge utilisateur, et ne modifie pas l'horloge de l'hôte. Ainsi, il ne nécessite aucun droit d'administration particulier, et n'interfère pas avec d'autres processus qui pourraient utiliser l'horloge de l'hôte.

Certaines constantes de temps ont été adaptées, notamment l'intervalle maximum entre deux requêtes de synchronisation, qui a été réduit pour améliorer la précision au détriment de la bande passante, et pour permettre de détecter plus rapidement une potentielle défaillance de l'horloge maître et d'élire une nouvelle horloge. Une défaillance du maître est typiquement détectée en moins d'une dizaine de secondes, ce qui est suffisant pour que la dérive des horloges clientes durant l'absence du maître reste faible.

### 3.5. Publication / Souscription

En s'appuyant sur le module de découverte de services, nous avons implémenté l'un des mécanismes d'interaction proposés en sous-section 2.1, à savoir un système de passage de messages par publication/souscription. L'échange de messages est organisé autour de la notion de *canal* : tous les processus abonnés à un canal sont notifiés lorsqu'un processus publie une information dans ce canal.

Les processus qui veulent souscrire à un canal déclarent un service de type `subscriber` auprès du module de découverte de service, en utilisant un nom qui identifie le canal utilisé. Pour publier un message OSC dans un

canal, un processus demande une liste des services de type `subscriber` ayant le nom du canal souhaité, puis envoie son message à chaque service de cette liste. Pour éviter de filtrer toute la table des descripteurs de services à chaque publication, la liste des abonnés au canal est mise en cache, et actualisée uniquement lorsque de nouveaux descripteurs sont ajoutés, ou que des descripteurs obsolètes sont supprimés.

### 3.6. Bibliothèque et *bindings*

Le prototype que nous avons implémenté prend la forme d'une bibliothèque écrite en C, qui comprend les modules suivants :

— un module de sérialisation et de désérialisation de messages OSC ;
— le module de découverte de services ;
— le module d'élection de maître ;
— le module de synchronisation d'horloge ;
— et le module de publication/souscription.

Un objet Max/MSP `osc_endpoint` permet d'exposer les fonctionnalités de publication/souscription de la bibliothèque dans cet environnement. Cet objet peut recevoir et envoyer des messages OSC sous la forme de messages Max possédant une syntaxe *ad hoc*, ou dans le format `FullPacket` utilisé par les objets de la bibliothèque Odot [21]. Un delai peut être spécifié lors de l'envoi d'un message par le biais de l'objet `osc_endpoint`. Dans ce cas, le message OSC est envoyé immédiatement, mais ne sera visible en sortie des objets `osc_endpoint` correspondants qu'à cette date. Il est donc possible d'absorber la gigue du réseau en envoyant des messages en avance tout en gardant une cohérence temporelle forte dans la réception et le traitement de ces messages. La bibliothèque est également accompagnée de *bindings* Python exposant la plupart de ses fonctionnalités.

A titre d'exemple, nous avons entamé une collaboration avec le compositeur Pedro Garcia-Velasquez, qui travaille sur une pièce pour orchestre de robots percussionnistes. Un essai préliminaire pour cette pièce peut être visionné en ligne [2].

Chaque robot est constitué d'un bras articulé tenant une baguette ou une mailloche, et contrôlé par un Raspberry Pi. Les gestes du robot sont définis dans un script python. La partition est lue de manière centralisée par un patch Max qui adresse les différents robots et synchronise leurs gestes. Sans notre bibliothèque, il serait nécessaire de régler manuellement les adresses IP et les ports de communication utilisés par le *patch* et par chacun des robots, et ceci chaque fois que la configuration du réseau change. De plus, il serait difficile d'absorber la gigue du réseau, ce qui conduirait à une distorsion rythmique de la séquence sonore produite par les robots. L'utilisation de notre bibliothèque à travers ses *bindings* Max et Python permet de résoudre ces deux problèmes de manière élégante, sans nécessiter l'installation ou la configuration d'une infrastructure supplémentaire.

[2] `youtu.be/bEXTr5SW8Y4`, accédé le 20 octobre 2020

## 4. CONCLUSIONS

Nous avons rappelé en section 1 l'importance des interactions temporelles dans le spectacle vivant, et nous avons montré qu'il existe un besoin pour une architecture permettant d'interconnecter les différents outils logiciels utilisés dans ce contexte, au travers d'un *middleware temporel*. Nous avons ensuite décrit en section 2 quelques composants essentiels d'un tel *middleware*. Enfin nous avons proposé une implémentation de ces composants en section 3.

Les prototypes que nous avons développés donnent des résultats encourageants. Notre bibliothèque OSC permet de composer et d'analyser des messages étendus pouvant représenter des structures hiérarchiques complexes, avec une performance meilleure que les alternatives existantes. Notre système de synchronisation d'horloge est le premier à notre connaissance à intégrer les algorithmes de filtrage et de mitigation de NTP dans une implémentation autonome destinée à l'informatique musicale. S'ajoute à cela les capacité d'élection d'horloge maître et de découverte de services, qui permettent de délivrer une solution « clé en main », ne nécessitant que peu d'infrastructure et de configuration préalable.

Nous pensons que les éléments présentés ici peuvent fournir les fondations d'un écosystème plus vaste incluant d'autres mécanismes d'interaction, tels que des appels de procédures distantes, des structures de données partagées, des points de rendez-vous et des barrières de synchronisation, permettant de couvrir une large palette de relations temporelles entre processus multimédia. Ces mécanismes pourront à leur tour servir de support à un système de détection et de réaction à des motifs temporels distribués, étendant la notion de motifs temporels introduite dans [12]. Nous espérons que cette approche permettra de répondre de manière modulaire et extensible aux attentes d'artistes portant une attention particulière aux interaction temporelles, et encouragera de nouvelles explorations dans le domaine de la co-création humain–machine.

## 5. REMERCIEMENTS

Nous tenons à remercier chaleureusement Jean Bresson et Jean-Louis Giavitto pour leur relecture avisée et leur précieux conseils.

## 6. RÉFÉRENCES

[1] Albright, S., Leach, P. J., Gu, Y., Goland, Y. Y., Cai, T. « Simple Service Discovery Protocol/1.0 ». Internet-draft, Internet Engineering Task Force, 1999. [`tools.ietf.org/html/draft-cai-ssdp-v1-01`, accédé le 20 octobre 2020.]

[2] Apple. « Bonjour overview », 2013. [En ligne ; accédé le 20 juillet 2019].

[3] Bencina, R. « oscpack », 2013. [En ligne ; accédé le 20 juillet 2019].

[4] Bray, T. « The JavaScript Object Notation (JSON) data interchange format ». Rapport technique, 2017.

[5] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F. « Extensible Markup Language (XML) 1.0 (fifth edition) ». Rapport technique, W3C, 2008. [En ligne ; accédé le 19 juillet 2019].

[6] Celerier, J.-M., Baltazar, P., Bossut, C., Vuaille, N., Couturier, J.-M., Desainte-Catherine, M. « OSSIA: Towards a unified interface for scoring time and interaction ». Proceedings of the International Conference on Technologies for Music Notation and Representation, Paris, 2015.

[7] Cheshire, S., Krochmal, M. « DNS-Based Service Discovery ». Rapport technique, 2013.

[8] F. Cristian. « Probabilistic clock synchronization ». *Distributed Computing* 3/3 (1989), p. 146-158.

[9] QLab. « QLab 4, User Reference Manual », Figure 53, 2020. [En ligne ; accédé le 12 mai 2020].

[10] Freed, A., Schmeder, A. « Features and future of open sound control version 1.1 for nime ». Proceedings of the New Interfaces for Musical Expression Conference, Pittsburgh, États-Unis, 2009.

[11] Friberg, A. , Sundberg, J. « Perception of just noticeable time displacement of a tone presented in a metrical sequence at different tempos ». *Journal of the Acoustical Society of America* 94/3 (1993).

[12] Giavitto, J.-L., Echeveste, J. « Real-time matching of Antescofo Temporal Patterns ». Proceedings of the International Symposium on Principles and Practice of Declarative Programming, Canterbury, Royaume-Uni, 2014.

[13] Google. *Protocol buffers language guide (proto2)*, 2008. [En ligne ; accédé le 19 juillet 2019].

[14] Gusella, R., Zatti, S. « The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd. » *IEEE Transactions on Software Engineering* 15/7 (1989), p. 847-853.

[15] Haas, H. « The influence of a single echo on the audibility of speech ». *Journal of Audio Engineering Society* 20/2 (1972), p. 146-159.

[16] Harris, S. ,Sinclair, S. liblo : lightweight OSC implementation, 2004. [En ligne ; accédé le 20 juillet 2019].

[17] IEEE. « IEEE Standard for a precision clock synchronization protocol for networked measurement and control systems ». *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, 2008.

[18] Jack, R. H, Mehrabi, A., Stockman, T., McPherson, A. « Action-sound Latency and the Perceived Quality of Digital Musical Instruments: Comparing Professional Percussionists and Amateur Musicians ». *Music Perception* 36/1 (2018), p. 109-128.

[19] Rabin, J. « libossia Documentation ». [En ligne ; accédé le 12 mai 2020].

[20] Kim, J. L., Belford, G. G. « A distributed election protocol for unreliable networks ». *Journal of Parallel and Distributed Computing* 35/1 (1996), p. 35-42.

[21] MacCallum, J., Gottfried, R., Rostovtsev, I., Bresson, J. , Freed, A. « Dynamic message-oriented middleware with Open Sound Control and Odot ». Proceedings of the International Computer Music ConferenceDenton, États-Unis, 2015.

[22] Matuszewski, B. « Soundworks - A Framework for Networked Music Systems on the Web - State of Affairs and New Developments ». Proceedings of the Web Audio Conference, Trondheim, Norway, 2019.

[23] McCallum, J. libo, 2015. [En ligne ; accédé le 6 août 2019].

[24] McCurry, M. « Rtosc - Realtime Safe Open Sound Control messaging ». Proceedings of the Linux Audio Conference, Berlin, Allemagne, 2018.

[25] Medialon Ltd. « Medialon Control System, User Reference Manual », 2019. [En ligne ; accédé le 20 janvier 2020].

[26] Metta, G., Fitzpatrick, P., Natale, L. « YARP: Yet Another Robot Platform ». *International Journal of Advanced Robotic Systems* 3/1 (2006), p. 43-48.

[27] MIDI Manufacturers Association. « The complete MIDI 1.0 detailed specification ». Rapport technique, 1982.

[28] Mills, D. « Network Time Protocol ». Rapport technique, 1985.

[29] Mills, D. *Computer network time synchronization - the Network Time Protocol*. CRC Press, Cleveland (OH), 2006.

[30] Mills, D. « Simple Network Time Protocol (SNTP) version 4 for IPv4, IPv6 and OSI ». Rapport technique, 2006.

[31] Mills, D., Burbank, J., Kasch, W. « Network Time Protocol version 4: protocol and algorithms specification ». Rapport technique, 2010.

[32] Mockapetris, P. « Domain names - implementation and specification ». Rapport technique, 1987.

[33] NetBIOS Working Group. « Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods ». Rapport technique, 1987.

[34] Nunes, D. S., Zhang, P., Silva, J. S. « A survey on human-in-the-loop applications towards an internet of all », *IEEE Communications Surveys & Tutorials* 17/2 (2015), p. 944-965.

[35] OASIS. « OASIS Web Services Dynamic Discovery (WS-Discovery) version 1.1 ». Rapport technique, 2009. [En ligne ; accédé le 20 juillet 2019].

[36] Object Management Group. « Data Distribution Service (DDS) ». Rapport technique, 2015. [En ligne ; accédé le 28 février 2019].

[37] Object Management Group. « Interface Definition Language version 4.2 ». Rapport technique, 2018. [En ligne ; accédé le 18 juillet 2019].

[38] Open Source Robotics Foundation. « Robot Operating System (ROS) », Rapport technique, 2010. [En ligne ; accédé le 28 février 2019].

[39] Wright, M. « Open Sound Control 1.0 specification », Rapport technique, 2002.

*Texte édité par Corentin Guichaoua.*