OSYNC: CROSS-APPLICATION PERFORMANCE SYNCHRONISATION OF BEAT-DRIVEN MUSICS

Alex Mesker

Macquarie University
Department of Media, Music,
Communication & Cultural Studies

Sarah Keith

Macquarie University
Department of Media, Music,
Communication & Cultural Studies

ABSTRACT

OSynC is a set of standardised messages for communicating metrical information and rhythmic descriptors for use in networked computer-based music performance. It uses an Open Sound Control (OSC) encoding to send descriptors from a host machine to any number of receivers, while receivers can in turn send changes to the host. The OSC protocol is an emerging standard for musical control, and is an extensible, low-bandwidth method for communicating information independent of platform and application. OSynC places particular emphasis on communicating rhythmic musical information at a range of musical timescales, providing details on musical context that are desirable for performance of beat-driven musics. It also aims to provide flexibility for musical improvisation, as well as intelligibility for performers. The authors describe OSynC's implementation as a Max patch, VST plug-in, and Max For Live device. These implementations are available from http://x37v.com/x37v/osync/.

1. INTRODUCTION

Collaborative computer-based music performance encompasses a wide variety of musical styles and configurations. From the laptop orchestra to smaller ensembles, a common requirement is the synchronisation of participants for precisely timed output. This is particularly crucial in the performance of beat-driven musics, including contemporary electronic dance music, such as IDM, dubstep, and so on. A system for networked performance of beat-driven musics must necessarily be robust and low-latency. Additionally, the system should provide the ability for real-time flexibility and improvisation in performance. The issue of synchronisation in performance is thus not purely computational, but also needs to take into account musical aspects that are most useful to performers.

Existing models for synchronisation may rely on non-computational means (i.e. initiating a simultaneous performance/clock start via visual or aural cues between performers), however different applications, computers, and hardware may suffer from varying latencies and jitter [1], causing eventual desynchronisation. Metronome-based timing information may also be sent across a network from one user to another, in the form of pulses or 'ticks', while timecode (or SMPTE) information can also be used as a means to indicate temporal position. The system proposed here (OSynC) aims to extend these existing paradigms. It describes a method for transmitting musical timing features that are important to performance, while maintaining flexibility and cross-application functionality. Given the continued growth of the laptop as performance instrument, networked synchronisation is not a new issue by any means, but remains an area for improvement.

2. EXISTING MODELS

To date, several methods have been used for synchronising two or more performers across a network. Early examples include MIDI clock and word clock, which are subject to comparatively significant limitations of bandwidth, distance, and frequency, and are prone to time drift. Other MIDI-based synchronisation methods include DIN sync, used for linking hardware synthesisers. MIDI, however, possesses a comparatively small bandwidth of 31.25 kilobits/sec, compared with 10+ megabits/sec for networking technologies used by Open Sound Control (OSC) [2]. Meanwhile, the frame resolution of MIDI timecode implies suitability for video rather than music, making it unsuitable for control-rate musical interactions. A further drawback to each of these methods is their tailoring towards hardware (rather than software) synchronisation.

Synchronisation using the computer's audio capabilities has been proposed as a solution to the limitations of MIDI. Max's *sync~* object, for instance, is designed to receive MIDI real-time pulses and fill in the 'blanks' to create an audio-rate phasor wave synchronised to a pulse. Streaming audio-rate information across a network, however, raises its own problems with regard to different machines possessing different clocks and sample rates. Although a number of ways for transmitting audio over a network have been proposed (such as Olaf Matthes' largely abandoned *netsend~* and *netreceive~* objects for Max and Pd [3], and Plasq's Wormhole2 VST plug-in [4]),

latency and synchronisation issues are not necessarily resolved with higher bandwidth. Furthermore, both of the above synchronisation systems lack the ability to transmit descriptive tags useful for musical performance.

There are a number of industry-supported methods for inter-application (and inter-machine) synchronisation. Both ReWire and VST SystemLink transmit playback-oriented information, including play state, bar, beat, and time signature information. ReWire, for instance, allows one application to drive another (client-server model), though this is not intended for use as a collaborative synchronisation tool for performers, as the assumption is that both programs are running locally. Meanwhile, Steinberg's VST SystemLink allows separate machines to be synchronised, but focuses primarily on sharing processor power between separate computers and additionally is developer-specific (i.e. both computers must be running Steinberg software). In 2008 Cycling '74 implemented a *transport* object for Max, designed to generate ReWire-like timeline-based transport information, but this is similarly intended for local machine communication within Max. The ability for inter-application communication is embedded in Max For Live, however communication is limited to Max and Ableton Live on a local machine.

OSynC differs from the systems described above, in that it is designed as a cross-application, control-rate method for synchronising sound objects in networked performance, which is platform- and application-independent. It focuses specifically on the synchronisation of musical attributes and their application in performance.

3. OPEN SOUND CONTROL

OSynC uses the Open Sound Control (OSC) protocol, an open system for network communication between computers and musical devices developed by the Center for New Music and Audio Technology (CNMAT), UC Berkeley [5]. The positives of an open communication protocol for musical control have been discussed in detail [6], in regard to applications such as communication of audio features [7], time-stamped clock synchronisation [8][11] and controlling VST plug-ins [9]. OSC's openness makes it a valuable tool for cross-application communication; as Wright [6] states, "Part of what makes OSC 'open' is that it comes with no standard set of messages every synthesiser must implement, no preconceptions of what parameters should be available or how they should be organised […] This form of openness has led to great creativity among OSC implementations, supporting idiosyncratic, creative software and hardware" [6, p. 194]. The OSynC standard aims to leverage the openness of OSC while proposing a base set of descriptors useful to musical performance and synchronisation.

OSynC proposes a set of standardised namespaces for synchronisation of rhythmic network-based performance. It also defines a priority of order for these messages. Similar OSC namespace standardisations have been proposed, such as SYN for synthesiser communication [10]. Wright [6] suggests that standardising namespaces is desirable for reasons of compatibility between applications and implementations.

OSC sends a continuous stream of 'packets' (or datagrams) describing the instantaneous state of a musical interface [11]. OSynC implements this idea, and reports information on the instantaneous state of position within rhythmic phrase on several timescales, as opposed to a metronomic pulse. Because OSC sends low-bandwidth packages via UDP (without an acknowledgement package sent in return from the receiving computer/s), OSynC messages can saturate the available bandwidth[1], or be sent at a user-specified rate. As OSynC sends a complete set of messages in each packet, dropped packets are not problematic, because the ensuing packets will contain the most up-to-date state information.

While research has been carried out on precise synchronisation using timestamps to allow systems to adjust for latency [8], OSynC does not timestamp messages for the sake of latency adjustment, although this could be provided in future implementations. It does however timestamp messages to ensure that message order is maintained. Clock synchronisation is not a concern in this research, as accuracy beyond the perception of the performer is not required[2]. Sample accuracy is therefore not necessary in OSynC, but a simple latency compensation can be implemented by timing a round-trip of packets and adjusting for latency on the host machine.

4. OSYNC

4.1. Description of OSynC

While OSC is typically used to send instrument and gestural control parameters and hardware control messages, OSC's openness as indicated by Wright [5] means that any musical information from the host can be treated as a transferable piece of information via OSC. Here, OSynC is used to transmit transport information between performers using a one-to-many host-receiver system. OSynC information is sent to receivers' IP addresses by the host via UDP. The host is responsible for transmitting OSynC messages, however any receiver of the system can alter the output of the host machine if desired (e.g. increasing master tempo). Although OSynC shares a number of similarities with ReWire's transport

[1] Tests show that between 13,500 and 17,000 datagrams per second could be sent between applications on a local machine, or between 3600 and 20,000 per second between performers over computer-to-computer network, depending on Max's Overdrive setting (benchmark tests carried out by authors). Speeds above this are possible, but may result in dropped packets.

[2] Wright [6, p. 195] addresses network latency, concluding that latency of up to 10ms is acceptable in a musical performance.

information, it possesses the advantage that the application does not need to support ReWire, only OSC. Furthermore, OSynC's descriptors are extendable to provide more detailed rhythmic information useful to performance.

4.2. OSynC information

OSynC transmits packets which define the following aspects of rhythmic state, in order:

0. Timestamp (for message order): integer
A low-resolution timestamp is included, not for latency adjustment or synchronisation, but to ensure that messages are received in the correct order. This safeguards against messages being re-ordered due to network load. If a message's time-stamp is earlier than the current message's time-stamp, it can be disregarded.

1. Play state (on/off): integer
Indicates whether the OSynC host transport is currently playing.

2. BPM/tempo: float
Indicates beats per minute as specified by the host.

3. Time signature: integer array
Numerator/denominator pair signifying number and type of beats.

4. Time elapsed between fractions (outlined below) as millisecond value: float
Used for synchronising time-based effects between users (such as delay times), or generating events above fraction-level resolution (e.g. at 120 BPM [4/4 time signature] the milliseconds between 32nd-note fractions is 62.5).

5. Total bar count from initialisation: integer
Indicates time (in bars) elapsed from the start of performance; can be used to schedule future events between performers.

6. Phrase-level bar count (1–4, or configurable by the user): integer
Allows performers to work within shared phrasings comprising multiple bars.

7. Beat within the bar (1–4 per bar, or configurable by the user based on time signature): integer
Allows performers to remain synchronised within the bar.

8. Fraction within bar (0–31 per bar, or configurable by the user based on time signature): integer
Indicates basic temporal framework for note event occurrence, with the assumption of 4/4 timing. Calculated based on time signature (32 × numerator ÷ denominator, e.g. 4/4 equates to 32 demisemiquaver events per bar; 3/4 equates to 24 per bar; 7/8 equates to 28 per bar). This representation has been used by the authors as a simple method of driving a step sequencer.

9. Ramp (0.–1. across the course of a bar): float
Indicates partial position within a bar, useful for creating arbitrary timing alterations on the receiver side.

A typical OSynC message packet will appear as follows:

```
/timestamp -773495082
/play 1
/bpm 125.7
/timesig 4 4
/fractime 59.665871
/barcount 12
/bar 4
/beat 3
/fraction 18
/ramp 0.578125
```

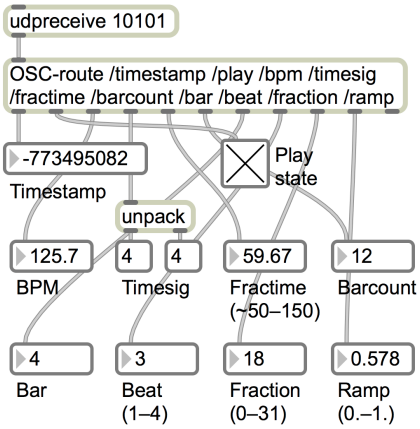

Figure 1. Received OSynC messages within Max

Early implementations of OSynC transmitted 'incomplete' packets (without timestamps) that only described changed attributes, hence only *ramp* messages were sent between *fraction* messages. It was therefore important that messages from the host machine were sent in the above order and that packets were not dropped. The current model for OSynC suggests that 'complete' messages should be sent to circumvent any ambiguity created by lost packets. Because of this single packet design, message order is less of an issue, but it is still recommended that the host packs the message in this top-down order to ensure that higher-level musical context-related information (e.g. *bar*) arrives before note-level instruction (e.g. *fraction*). Lastly, it should be emphasised that OSynC is particularly dedicated to performance use; its current

implementation forgoes sample accuracy in favour of flexibility and ease of use in a performance situation. Nonetheless, it is capable of synchronisation with no audible latency between performers.

4.3. Benefits of OSynC

OSynC therefore does not rely on the strict regularity of pulse, but rather the regular update of information. Jitter in regularity, or dropped packets, are not a major concern, given the high rate of messages sent. OSynC differs from existing synchronisation protocols such as MIDI real-time messages in that if a connection is lost, positional synchronisation with the host is re-established as soon as the next package is received, instead of continuing from the last dropped packet.

OSynC focuses on sharing useful musical information between performers on a variety of timescales, from the macro-level (*play* and *barcount*), to meso-level (*bar*), sound object level (*beat* and *fraction*), down to micro-level (*ramp*) (after Roads [12]). This allows performers to work with different levels of rhythmic and structural form, in a way that is intelligible to all participants. OSynC additionally allows bidirectional communication between host and receiver; although all OSynC messages originate from the host, the receiver can alter higher-level controls such as tempo, time signature, and play state, forming a discontinuous feedback loop (Figure 2).

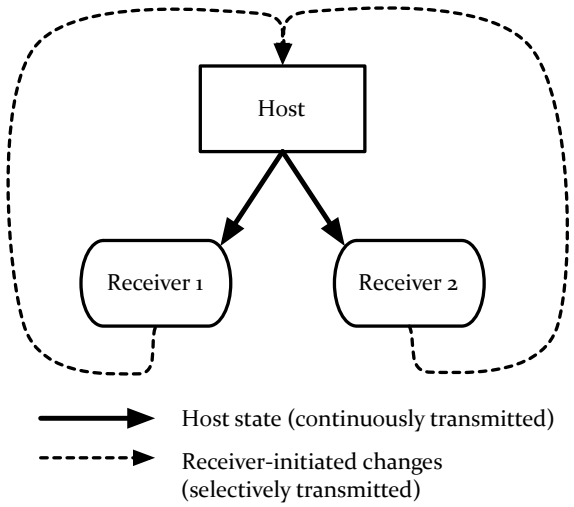

Figure 2. Host/receiver network flow using OSynC

A receiver can use OSynC in a prescriptive sense, or re-interpret transmitted values to their own ends (e.g. adding *ramp* value to *barcount* in order to provide continuous representation of position[3]). The *ramp* value, in particular, is a powerful musical descriptor. A receiver can use the *ramp* value as an arbitrary method of counting a number of beats within a bar by multiplying it by an integer or float, to create polyrhythmic interplay against the host to any degree of complexity. Additionally, *fractime* could be multiplied or divided to convey tempo-synchronous timing events beyond the fraction level.

Another benefit of OSynC is that host and receiver machines can run at different sampling rates and require no specialised hardware, whereas synchronisation systems such as VST SystemLink require sample rates to be the same in both systems. Additionally, OSynC is not software- or hardware-dependent, and can be implemented as a standalone within a development environment (Max, Pd, SuperCollider), or as a plug-in if non-development software is used as the host (e.g. multi-track audio software).

4.4. Current implementation

The first version of OSynC was built in 2007–2008 [13]. Its initial goal was to improve communication between the multi-track recording environment Cubase and MaxMSP, using Cubase to both maintain tempo and capture events generated in Max, while maintaining a uniform position between both programs to make subsequent editing easier. To do this, an OSynC host VST plug-in was built using the Pluggo SDK, which leveraged *plugsync~*'s ability to obtain ReWire-like transport information from a host environment. Here, Cubase generated timing events that were interpreted by MaxMSP, allowing synchronisation between the two applications on the local machine. One of the drawbacks of this implementation was that OSynC transmitted the host's state *once* per signal vector. The vector size used by plug-ins was the same as the I/O size, meaning that a VS of 512 samples produced up to 11ms waver, a limitation of this particular host environment. There was therefore a tradeoff between temporal accuracy and efficiency. This initial implementation also used a simplified descriptor system, which only updated *changed* values (i.e. *bar* was only reported at the start of a new bar). This contradicted OSC's remit as a stateless message system.

Since the introduction of Max For Live (2010), an implementation of OSynC within this multi-track recording environment is a simple task[4]. Similarly, the introduction of the *transport* object for Max enables Max to act as a host system by wrapping OSC descriptors around the *transport* object's output. Although these programs enable synchronisation, they do not in

[3] This could be useful in controlling elements that should evolve over the course of a musical performance, and could foreseeably be used to control non-musical aspects of a performance, such as lighting.

[4] The re-integration of many Pluggo objects into the Max For Live package means that a network synchronisation system like OSynC can be built in Ableton Live, using Live as the host and *plugsync~* to report on host state.

themselves suggest a solution to cross-application (beyond Max and Live) or networked synchronisation. A wider implementation of OSynC as a VST plug-in, for example, would allow any VST-compatible host to act as an OSynC host, and any implementation of OSynC within an audio development environment like SuperCollider, Max, or Pd to act as a host, or as a receiver. The current implementation of OSynC as used by the authors is as a means of synchronising two performers using Max as a generative performance environment, however there are myriad potential uses for the OSynC system. Implementations of OSynC for Max, Ableton Live, and Pluggo are available from http://x37v.com/x37v/osync/.

4.4.1. Example 1: Networked cross-application performance using a variable number of participants

A host running Ableton Live transmits OSynC information, via Max For Live, to two receivers across a network. One receiver is using Max, and another is using Pd. The Ableton Live host is triggering scenes at a variety of BPM and continually addressing both receivers using OSynC. As the scene is changed, each receiver is updated with current information in order to generate note events in time with the host. The receivers' time-synchronous effects (e.g. delay) are updated via *fractime*. Both receivers can additionally slow down or speed up the BPM of the Max For Live host. A third receiver, running Processing, creates time-synchronous visuals alongside the performers.

4.4.2. Example 2: Working with flexible timings in OSynC

A host, using an implementation of OSynC in SuperCollider, transmits timing information to a receiver running Max. Using transitions in the *ramp* value to generate note events, the Max receiver can work at a different time signature to the host in order to produce polyrhythmic output, while still staying synchronised with the host at the bar- and phrase- level.

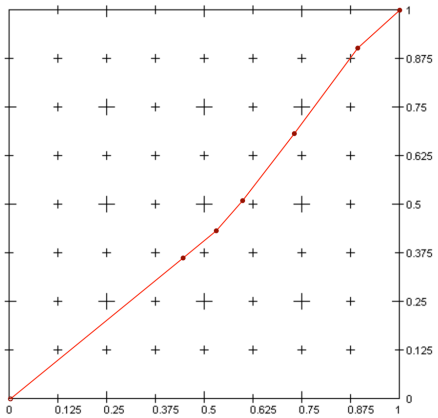

Figure 3. Altering linearity of playback using *ramp* value in conjunction with a lookup function

Secondly, the Max receiver can use a lookup function to create temporal variation [14] from the events output by the host at the bar- or beat- level, such as dynamic temporal phrasing and swing (Figure 3). Additionally, the Max receiver uses a combination of *bar* and *ramp* values to drive a dynamic filter effect over the course of a four-bar phrase.

4.5. Future implementations

The current implementation is limited to receivers known to the host, i.e. the host explicitly sends OSynC messages to fixed IP addresses. An area to investigate in future is the ability for a new receiver to join a multicasted performance by requesting OSynC packets from a host, without prior explication of the receiver's address. Further research into the scalability of performer numbers and network load would also be required. Additional musical metadata such as rhythmic phrasing, root note, and musical mode could also be implemented to allow performers to synchronise modal changes during a performance. Extended benchmarking of OSynC to increase efficiency is also a possibility. A more formalised rule for timestamping based on OSC timetags, to provide high-resolution timestamping of OSynC packets, is also an area for future inquiry.

5. CONCLUSION

Although OSynC has been used as a performance synchronisation system by the authors for a number of years, this paper proposes OSynC as a method for standardising beat-driven performance, rather than as a definitive implementation of this idea. Attention has been paid to prioritising useful performance information, while maintaining intelligibility and controllability. One of the benefits of OSynC as proposed here is that descriptors can be used, ignored, or extended, according to performer needs. All musical factors needed by the authors are described here, however musicians may also wish to communicate additional information such as pitch spaces, timbral settings, dynamics, rhythmic groupings, light controls, text, and other information not specifically related to synchronization. With this alpha implementation these additional ways of participant synchronisation have been considered but have been left up to the end user. The authors welcome other implementations for other software.

The standardisation of an open synchronisation protocol encourages compatibility between applications and third-party plug-ins, as well as performers. Initial results from the use of OSC as a means for synchronisation show promise for wider application. Because OSC is an open protocol requiring no special hardware, and the number of programs supporting this

standard continues to increase, OSC (and OSynC) may be fertile ground for future development.

6. REFERENCES

[1] Innerclock Systems Pty Ltd 2012, 'Precision Midi Clock Din Sync and Tempo Synchronisation Solutions', online at: http://innerclocksystems.com/New%20ICS%20Clock%20Watch.html, viewed 24 February 2012

[2] Simpson, S, no date, 'Open Sound Control: an overview', online at: http://www.ixi-audio.net/content/info/osc.html, viewed 24 February 2012

[3] Matthes, O 2005, 'netsend~ for Max/MSP and Pure Data', online at: http://www.nullmedium.de/dev/netsend%7E/, viewed 24 February 2012

[4] Plasq 2007, 'Wormhole2—Audio plugin to transfer realtime audio over a network', online at: http://code.google.com/p/wormhole2/, viewed 24 February 2012

[5] Center For New Music and Audio Technology (CNMAT), no date, 'OpenSoundControl.org Introduction to OSC', online at: http://opensoundcontrol.org/introduction-osc

[6] Wright, M 2005, 'Open Sound Control: an enabling technology for musical networking', *Organised Sound* 10(3), pp. 193–200

[7] Schmeder, A, Freed, A & Wessel, D 2010, 'Best Practices for Open Sound Control', *Proceedings of Linux Audio Conference*, Utrecht, Netherlands, online at: http://opensoundcontrol.org/files/osc-best-practices-final.pdf, viewed 24 February 2012

[8] Dannenberg, R 2004, 'Clock Synchronization for Interactive Music Systems', *Proceedings of OSC Conference 2004*, Berkeley, USA

[9] Zbyszynski, M & Freed, A 2004, 'OSC Control of VST Plug-ins', *Proceedings of OSC Conference 2004*, Berkeley, USA

[10] Ehrentraud, F 2007, 'SynOSCopy', online at: https://github.com/fabb/SynOSCopy/wiki, viewed 24 February 2012

[11] Schmeder, A 2008, 'Everything you ever wanted to know about Open Sound Control', online at: http://opensoundcontrol.org/publication/everything-you-ever-wanted-know-about-open-sound-control, viewed 24 February 2012

[12] Roads, C 2001, *Microsound*, MIT Press, USA

[13] Mesker, A 2008, 'OSynC – Synchronising applications with a VST Plugin', online at: http://x37v.com/x37v/writing/osync-synchronising-applications-with-a-vst-plugin/, viewed 24 February 2012

[14] Schacher, JC & Neukom, M 2007, 'Where's the beat? Tools for dynamic tempo calculations', *Proceedings of the 2007 International Computer Music Conference,* Copenhagen*,* pp. 17–20