Rhythmic Synchronization of Events based on OSC Data from an External Source

Øyvind Brandtsegg

Norwegian University of Science and Technology
oyvind.brandtsegg@ntnu.no

Abstract. OSC messages is an efficient and versatile manner in which to communicate between Csound and other software. An inherent drawback of using network communications is the potential for timing jitter. An event displacement of just a few a few milliseconds will in many cases be perceived as a problem for music performance. To alleviate these potential problems, one can use different methods for time stamping each OSC message, and the receiving module can use this for rhythmically precise synchronization when playing the events. The current article explores a method for such rhythmical synchronization within Csound.

Keywords: OSC, Python, Rhythm, Timing and Synchronization

1 Introduction

Csound is an extremely versatile audio synthesis system, with its large number of specialized opcodes. Recent developments of the language has also facilitated its use for complex event generation (for example [1], [2] and [3]). For some use cases, it can be useful to take advantage of data processing in other programming languages. A number of interfaces are available for this purpose, most significantly the Csound API that allows Csound to be embedded as a module in many popular programming languages ([4] and e.g. [5]). There are also facilities for running snippets of other programming languages inline in Csound code, for example Python (the py opcodes[1]), Lua, and Faust[2]. These technologies allow a close integration with Csound and many times this is desirable. In some other cases we might need to have a more compartmentalized architecture, with clear divisions between components in the system. This can allow for system modules that each are relatively simple, with less dependencies([3]) which can alleviate potential version conflicts when upgrading system components, and cleaner separation of processing threads. In such a compartmentalized architecture, we can use a generic method of communication and signaling between modules. For this purpose, Open Sound Control (OSC) [6] provides a viable protocol. Since OSC uses generic network connection, it also allows the system to be split over several

[1] https://csound.com/docs/manual/pycall.html
[2] https://csound.com/docs/manual/faustdsp.html
[3] even though the system as a whole still has a large set of dependencies

computers if needed. One inherent drawback of using network communications is the potential for timing jitter. The amount of jitter depends on the quality of the network connection and the available bandwidth, but it can typically be of an order that can disturb rhythmic precision in music performance. An event displacement of a few milliseconds will be perceived as a problem for music performance in many cases. With regards to the acceptable bounds on timing jitter, Friberg and Sundberg [8] reports that a timing displacement of 6 ms is perceivable in a monotonic isochronous sequence. However, the sequences measured used inter-onset times (IOT) in the range 100 ms and upwards. Fujii el al. [9] reports that professional drummers can achieve a mean synchronization error of 1-2 ms. Even though it is hard to determine any absolute value for acceptable timing jitter, in our system we can aim for a performance accuracy on par with that of a professional drummer. For algorithmic computer music, we might envision creating very fast rhythmical patterns, unplayable by a human performer, sometimes crossing over from the domain of rhythm to the domain of perceived pitch. This happens for example in granular synthesis [10]. In audio synthesis, phase relations between different simultaneous sound sources can have significant perceptual effects. This means that an acceptable jitter lies in the range of the audio sample rate (e.g. 0.02 ms at a sampling rate of 48kHz). Ideally, the timing precision of a flexible algorithmic rhythm generator should then be accurate to the audio sampling rate in order to allow correct reproduction of musical ideas transcending the rhythm to pitch time strata. To alleviate timing jitter problems, one can use different methods for time stamping each OSC message, and the receiving module can use this for rhythmically precise synchronization when playing the events. This method has also been described by Dannenberg [11], trading jitter for latency. The current article explores a method for sample accurate rhythmical synchronization of OSC generated events within Csound. The article refers to a set of example scripts with implementations of the methods discussed. These scripts should be available from the same source where the article was found and also at a github repository[4]. Specific scripts are referenced by file name in the text, like this: *myfile.csd and myfile.py*

1.1 Use Case

The use case for the system proposed in this article is realtime algorithmic composition with performer interaction. The system should be able to access data from an external source without interrupting the audio processing thread(s). If a chosen algorithm for data generation takes a long time to return, the audio system must continue processing at the regular pace, gracefully catching up with skipped events when data is available. This means that single events may be delayed or skipped, but that the rhythmic pulse(s) of the system does not drift. In this manner, polyphonic rhythmic processes can proceed without any voice drifting out of sync with another. One assumes that a composition algorithm suited for realtime performance will in most cases be able to deliver data for

[4] https://github.com/Oeyvind/rhythmic-sync-osc-csound

the next event within a relatively short time frame, but also that occasional processing burst might be necessary. As is common, we use the term *delta time* to describe the time from the current to the next event. The system must allow a delta time of zero, to also be able to stack events for simultaneous performance (chords and similar).

For the case of example, we will use Python to implement the data generating algorithm. The example algorithm used in this article will be very simple. For the purpose of utilizing just this algorithm, it would obviously be better to just implement it directly in Csound. Algorithm sophistication is not the focus of this article, but we try to outline a practical manner in which system components may be interfaced in order to allow more sophisticated and processor intensive algorithms.

It is preferable to be able to run the system in the context of a regular Digital Audio Workstation (DAW) environment, as this allows access and integration with off-the-shelf music software. With the plethora of both paid and free available sound generators, effects, production tools and utilities, it seems unreasonable to aim for a system that closes out those possibilities. As we will want to run Csound within the context of a DAW, we will design the system so that the Csound part of it can be embedded in a VST (using Rory Walsh's Cabbage [12]). Since it is Csound/Cabbage that acts as the main program, we can not use Csound API to communicate with Python (then Python would need to be the main program). Further, we will refrain from using the Python opcodes in Csound, as that would mean we have Python processing happening inside the audio processing thread. We would like to keep the audio processing thread as light as possible, also because any interrupts within the audio loop of a VST plugin will adversely affect the audio processing of the whole DAW host application. For the sake of simplicity, we will still run Csound as a standalone application in the context of this article, so we don't get entangled in the design of a GUI and other niceties. This also opens up the proposed system design to other applications and use cases.

2 System design considerations

2.1 Basic system with relaxed timing constraints

The basic system is quite simple: Csound runs a timed process asking Python for data for the next event, and creates the event when data is received. This has some problems that we will look into, but as an overview of what we try to achieve, we can look at figure 1.

In this most basic example implementation (*basic_osc.csd and basic_osc.py*) we use a metro opcode in Csound to trigger the data request over OSC, generate a note number in Python and send that back to Csound over OSC. A note event is generated as soon as the message arrives in Csound. This is method will work well in many cases where timing and synchronization is relaxed. For example control parameters for an algorithm, automation of instrument parameters like volume,

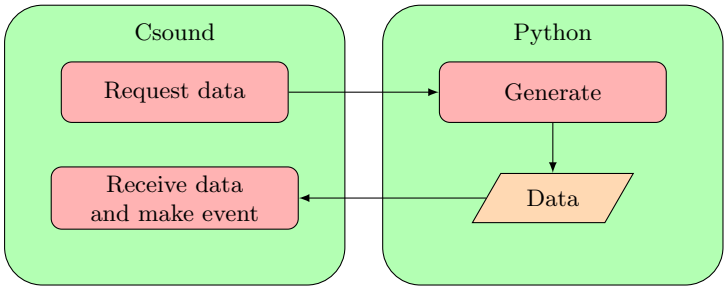

Fig. 1. Basic overview of system

effect sends etc. However, the rhythmic precision of the performance suffers from network timing jitter. In this implementation, we use a steady metro in Csound to trigger the data request, so the timing will not drift over time even if individual messages might be delayed. If we want to allow more complex rhythms, we might want to also request rhythm and tempo data (in the form of event delta time) from Python. If a tempo change is received late, the resulting rhythm will drift. For a polyphonic system, we want to ensure synchronization between separate voices, and make sure that the rhythm data and tempo changes are not affected by network timing jitter.

2.2 Implementation with rhythmic precision

Rhythmic synchronization can be achieved by collecting events into a queue and then dispensing them at the appropriate time using a master clock in Csound. Then we can also add a lag time to the queue dispenser as necessary to receive all events in due time. This requires a flexible data container to hold the event queue, a container that can mix different data types and hold events with varying numbers of parameters. Storing, parsing, and updating the queue could quickly become a complex task. Interactive performance might also require refreshing and/or deleting events belonging to a specific voice without affecting other voices, and we might want to cancel future events according to some constraint. *But wait, we already have that in Csound!* The internal scheduling mechanism that reads score events, and also allows insertion of events in realtime with a specific event delay. All data types that can be accepted as instrument event parameters are already taken care of. The timing of the scheduler is sample-accurate and also allows deletion of future events[5]. Figure 2 shows an overview of the system, using a master clock in Csound with event delta times for synchronization. This represents an overview of the implementation done in *precise_timing.csd* and the corresponding .py file. The example implementation there shows a polyrhythmic passage where this tight synchronization is needed. Listing 1 and 2 respectively shows code excerpts for sending and receiving OSC data.

[5] using the turnoff3 opcode

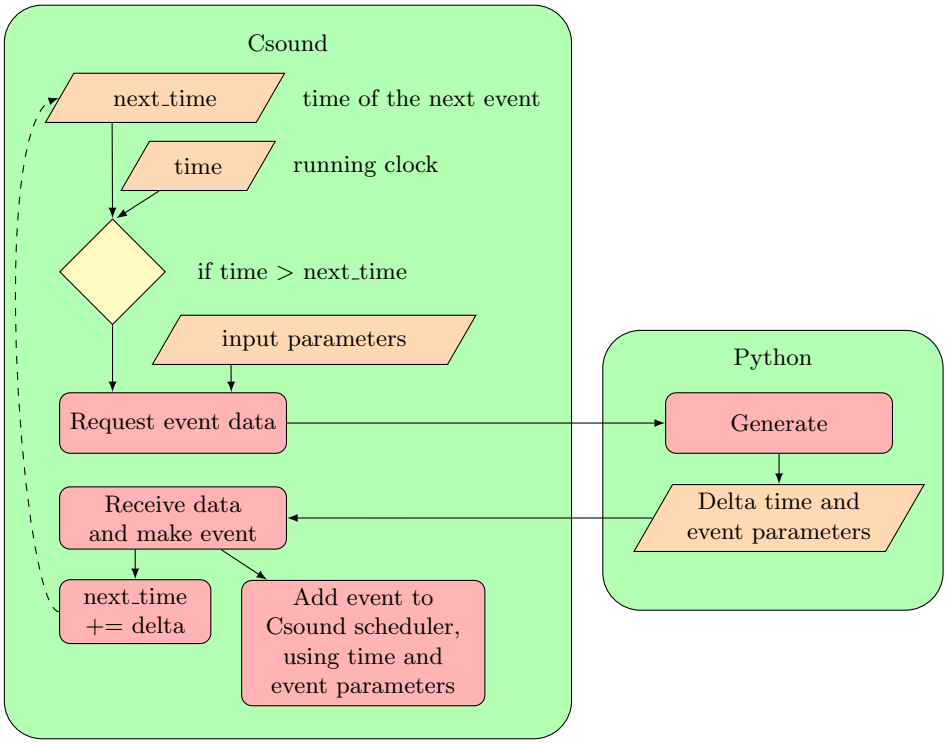

Fig. 2. Timing and event generation

Data request

```
kget_event = (ktime > knext_event_time) ? 1 : 0
if kget_event > 0 then
OSCsend kindex+1, "127.0.0.1",9901, "/csound_send", \
  "iifi",ivoice, kindex, itempo, ibasenote
endif
```

Listing 1: Excerpt from the file *precise_timing.csd* showing the request for event data

Receive data and process event

```
kmessage OSClisten gihandle, Saddress, \
  "ifff",  kindex, kdelta_time, knote, kduration
if kmessage == 0 goto done
kevent_time_delay = (knext_event_time-ktime) + ievent_trig_lag_time
event "i", 51, kevent_time_delay, kduration, iamp, knote
knext_event_time += kdelta_time
```

Listing 2: Excerpt from the file *precise_timing.csd*; Receive event data and insert event in Csound scheduler

In our example here, we just read values off a list, but we can conceivably use a data generating algorithm that takes more time to compute its values. To facilitate this, we can use a larger static lag time for event synchronization in Csound. Better still would be to let the algorithm generate lists of values in larger chunks and then trigger generation of a new batch of values when the previously generated list of values is almost consumed. One could trigger such processing when the last, or next-to-last value is dispatched, and then have ample processing time to generate new values. The generating algorithm would then still be reasonable up to date with external variables (e.g. what is happening in other simultaneous voices, or actions taken by the performer).

Even if simple serial reading of values from lists is a somewhat limited data generation algorithm, it can have some musical applications. A short composition (*serial_composition.csd and serial_composition.py*) has been included in the example scripts, showing a slightly more musical use of serial data for pitch, rhythm, stereo panning, attack time and reverb send levels.

2.3 Embedding as a plugin for use with a DAW

The proposed implementation allows embedding in a plugin architecture (i.e. VST) for use within a DAW environment. An implementation of a VST version is included in the examples. It shows a GUI controlling 2 separate voices that can be started and stopped individually with a play button. They will stay in sync with the timing relationship with which they were started. Each voice has individual data series for pitch and rhythm, and the data can be updated from a text box in the GUI. This is intended as a proof of concept, and made relatively simple for the case of clarity.

3 Running the provided code examples

The code examples for this paper consist of pairs of files with the same file name. A Csound file with the csd extenstion and a Python file with the py extension makes up such a pair. To run them, open two terminal windows. First run the Python file, it act as a server, waiting for data requests from Csound. Then run the Csound file, which will request data from Python and generate sound. The Csound process will terminate after the score is done, but the Python process must be terminated manually. The Python script requires the pythonosc library [13]. The VST example can be compiled with Cabbage [12]. In this case one should start the Python server script at any time before hitting "Play" in the VST GUI.

4 Conclusion

An example implementation has been shown that allows sample-accurate algorithmic event generation in Csound based on data received via OSC from an

external data generator. Python was used as the data processing server, but other data sources would work similarly. A master clock in Csound was used for rhythmic synchronization, and the internal Csound scheduler was used to dispense events with sample-accurate precision. Several pairs of csd/py scripts may be run simultaneously, provided they each use separate network ports. It is also possible to allow several csd scripts to interact with one master py script. This could be useful to allow several VST plugins running this script to interact with a central realtime composition algorithm, where e.g. voice leading decisions might rely on currently active notes in other voices running on separate tracks in the DAW.

References

1. Lazzarini, V. et al. (2016). *Csound: A Sound and Music Computing System.* Springer.
2. Yi, S. *libsyi - Library of Csound UDO code.* `https://github.com/kunstmusik/libsyi`
3. McCurdy, I. *Realtime Csound Examples.* `http://iainmccurdy.org/csound.html`
4. *Csound API documentation* `https://csound.com/docs/api/index.html`
5. *Ctcsound API Examples* `https://github.com/csound/ctcsound/blob/master/cookbook/08-ctcsoundAPIExamples.ipynb`
6. *Open Sound Control Home Page* `https://cnmat.org/OpenSoundControl/`
7. Wessel, D. and Wright, M. (2002) *Problems and Prospects for Intimate Musical Control of Computers.* Computer Music Journal vol 26-3. `https://doi.org/10.1162/014892602320582945`
8. Friberg, A. and Sundberg., J. (1995) *Time discrimination in a monotonic, isochronous sequence.* The Journal of the Acoustical Society of America, 98(5):2524-2531, 1995.
9. S. Fujii, M. Hirashima, K. Kudo, T. Ohtsuki, Y. Nakamura, and S. Oda. (2011) *Synchronization error of drum kit playing with a metronome at different tempi by professional drummers.* Music Perception, 28(5):491-503, 2011.
10. Brandtsegg, Ø. and Saue, S. and Johansen, T. (2011) *Particle synthesis–a unified model for granular synthesis.* Proceedings of the 2011 Linux Audio Conference. `http://lac.linuxaudio.org/2011/papers/39.pdf`
11. Dannenberg, R. (1989) *Real-Time Scheduling and Computer Accompaniment.* In M. V. Mathews and J. R. Pierce, eds. Current Directions in Computer Music Research. Cambridge, Massachusetts: MIT Press, pp.225–261.
12. Walsh, R. *Cabbage - A framework for audio software development.* `https://cabbageaudio.com`
13. *Python OSC - Open Sound Control server and client implementations in pure Python.* `https://pypi.org/project/python-osc`