Composing the Network with Streams

Ian Clester
ijc@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, United States

ABSTRACT

We present Aleatora, an early-stage framework for building compositions from lazy, effectful streams. Aleatora's streams, which may be combined by *sequential*, *parallel*, or *functional* composition, are well-suited to expressing interactive and aleatoric musical compositions. Aleatora includes a networking module which aids in writing compositions for the Internet of Sounds using network data sources such as OSC, external APIs, and Internet sound repositories (e.g. Freesound). This paper describes the design and implementation of Aleatora and demonstrates how it can facilitates weaving external input sources, such as network streams, into compositions.

CCS CONCEPTS

• **Applied computing** → **Sound and music computing**; • **Human-centered computing** → **Interaction design**.

KEYWORDS

Composition, audio synthesis, music programming languages, stream-based programming, Internet of Sounds

ACM Reference Format:
Ian Clester. 2021. Composing the Network with Streams. In *Proceedings of Audio Mostly 2021: Sonic experiences in the era of the Internet of Sounds (AM'21)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Computers are fundamentally more powerful than preceding music playback technologies. Beyond replaying a fixed recording, repeating the exact waveform that was recorded by a microphone or put together in a studio, computers can replay the *steps* necessary to realize a composition. This suggests an expansion of the notion of composition, from a recorded waveform or a sequence of notes to a *program* that generates musical output—a generator for "a field of possibilities" rather than a single fixed outcome. [5]

Composers are still grappling with the possibilities. Since the dawn of digital computers, much progress has been made in the design and implementation of digital audio workstations (DAWs) and audio programming languages. However, these tools are essentially based around the timeline and signal-flow graph: abstractions that can each only solve part of the puzzle. Weaving interactive, aleatoric, or externally-influenced elements into a composition still presents a unique challenge.

In this paper, we propose Aleatora, a framework for musical composition built on an unusual abstraction: lazy, effectful streams. Streams represent a repeatable chain of computation, and they provide an elegant way for the composer/programmer to perform *sequential*, *parallel*, and *functional* composition in a consistent way with both fixed and dynamic materials. We briefly describe Aleatora's design and implementation and demonstrate how it simplifies working with network sources, enabling the composer to write them in to a structured composition and "compose the network."

1.1 Related Work

Recent work on the Internet of Sounds (or the related areas of the Internet of Audio Things and Internet of Musical Things) has focused on the instrument: Internet-enabled *smart instruments* that combine sensors, computational capabilities, and network connectivity.[7] For example, [8] describes a Smart Guitar capable of networking with other smart devices during performance. External influences (e.g. an app on a phone or streaming audio) can affect the guitar's output, and the performer's input can have external effects (e.g. controlling a DAW or a VR environment).

Our work is complementary, approaching from the opposite direction: we focus on how to effectively realize Internet-enabled compositions. Aleatora addresses the challenge of "composing the network," as articulated by Turchet et. al. in [9] — the need for "new forms of composition" for the Internet of Sounds, with tools for the composer "to support and control distributed, heterogeneous capabilities," and compositions capable of "recall and reproduction."

In the domain of audio programming languages, there are a variety of tools available. However, most of these[1] use the core abstraction of a signal flow graph, with a separate language or interface for imperative code to construct and manipulate the graph. Our work differs from most in this area in its core abstraction of streams (signals with potential endings) and the operations these support. This abstraction obviates the need for strict separations between score and orchestra *and* synthesis and control, and it makes it easier for the composer to compose horizontally as well as vertically. (This point is elaborated in section 2.1.)

Nyquist [4] is Aleatora's closest conceptual relative. Aleatora shares with Nyquist several crucial features, including an interactive environment based on a dynamic language with a REPL,[2] a lack of distinction between score and orchestra, sounds as first-class

AM'21, September 01–03, 2021, Trento, Italy

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] e.g. Max/MSP, Pure Data, SuperCollider, Csound, ChucK, JavaScript with Web Audio
[2] That is, a read-eval-print-loop, or interactive language shell

values, streams ("signals" in Nyquist) as potentially-infinite lazy linked-lists, and support for arbitrarily mixed sample rates.

That said, our work differs from Nyquist in a few essential ways. [3] Aleatora supports composing with external events (e.g. TCP data, OSC or WebSocket messages, tweets) by conceptualizing them as ("impure" or side-effectful) streams of data, treated the same as any other stream. It does not memoize[3] streams by default, so the computation may take a different path and yield a different sequence the next time through (due to side-effects). It provides convenient syntax for common operations by overloading operators in the host language. And it is implemented as a library for a popular general-purpose language rather than taking the form of a domain-specific Lisp dialect. (And, it must be said said, Aleatora is not yet as mature and complete as Nyquist and does not deal with some issues such as block computation, logical start and stop times, and behavioral abstraction for context-dependent transformations.[4])

Looking further afield, to computer science and programming languages generally, there is important related work on the stream abstraction. The "Streams" section of Abelson and Sussman's *Structure and Interpretation of Computer Programs* [1] is a classic reference. The ideas are refined in SRFI-41 [2], a common extension to the Scheme programming language. In both SICP and SRFI-41, streams are memoized, and mixing streams with side effects is discouraged. This is similar to Haskell's native List type, which is naturally lazy and prohibits side effects. In contrast, Haskell's `streaming` library [5] offers a streaming abstraction that elegantly combines laziness and effects. `streaming` makes use of Haskell's robust static type system and offers complexity beyond what we consider here, but it shares Aleatora's core abstraction and has served as a design inspiration.

2 DESIGN & IMPLEMENTATION

2.1 Types of Composition

First, we elaborate on the three kinds of composition Aleatora supports. As illustrated in 1, these are *sequential*, *parallel*, and *functional* composition. Many systems support two of these well, and may have some support for the third tacked on, but first-class support for all three is rare.

Sequential composition entails concatenating two processes: following one process with another, such that the combined output is the output of the first followed by the output of the second. In general-purpose languages, this is often the ; operator or the newline. Musically, sequential composition corresponds to playing one thing after another, splicing tape, placing one clip to the right of another in a DAW timeline, or writing one note after another in Western music notation.

Parallel composition entails running two processes at the same time.[6] For example, in Pd or MSP, the user can compose two patches in parallel by connecting them both to the `dac~`; as in hardware, things run in parallel by default. Each subpatch continues to execute

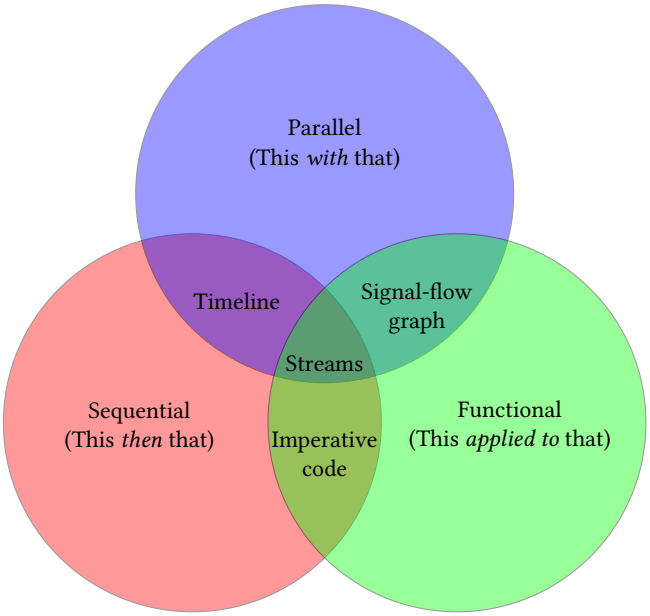

Figure 1: Types of composition and the abstractions that support them.

independently, and their outputs are summed at the output port. Musically, parallel composition corresponds to playing two things at the same time, placing one clip on top of another in the DAW timeline (on another track), or writing one note on top of another (as in a chord, independent voices, or multiple staves) in Western music notation.

Functional composition entails transforming the output of one process by another process. This is akin to linking two guitar effects pedals together with an audio cable; linking two objects in Pd or MSP with a virtual cable; calling `AudioNode.connect()` in the WebAudio API; or using the `=>` operator on unit generators in ChucK. In Western music, this is partly represented by notation that affects a performer's interpretation (dynamics, tempo markings, instructions like "pizzicato," "vibrato," etc.) and partly carried out by the composer manually (as in sequencing a melodic fragment by transposing it).

Each pair of two kinds of composition corresponds to a familiar abstraction. Parallel + functional composition are the essence of the signal flow graph, and they are well-supported by practically all audio programming languages. They arise naturally from how circuits are constructed and thus are the core of analog and modular synthesis. Sequential + parallel composition are the essence of the timeline (with horizontal time and vertical layers), and they are well-supported in multitrack tape, or its digital equivalent in the DAW. Sequential + functional composition are the essence of purely imperative code (as in a sequence of instructions, in a single thread/process, generating samples).

This correspondence also implies that each of these familiar abstractions is missing something. The timeline is missing functional composition, so DAWs augment it by supporting plugins—something *outside of* the timeline. The signal flow graph is missing

[3] A memoized stream is one that stores computed results, yielding the same sequence of values the next time. This is a common stream optimization, but if streams have intentional side-effects or nondeterminism, it may be undesirable.
[4] Although the latter can be attained in part through operator/method overloading.
[5] https://hackage.haskell.org/package/streaming
[6] Note that this does not actually require that the processes run in parallel (hardware parallelism), just that both processes can contribute to the same samples in the output.

sequential composition, so most audio programming languages augment it with an imperative language that can manually manipulate the graph. The imperative code can be sequenced, but this does not actually concatenate the output; the execution of the audio graph is ultimately outside the user's control.

2.2 Streams

Streams support all three kinds of composition. A stream represents a sequence of values generated on-demand. Streams operate by returning a tuple of (next element, rest of stream) or a Return value, which indicates that there are no further values.[7] Streams do not generally mutate when they are called, so they can be played back from any earlier point (assuming the user kept a reference to it). Since streams are not memoized by default, the second playback may produce a different sequence of results than the first.

The ability to end may appear to be an additional complication of streams, as compared to signals, which do not end but may simply dwindle to zero (silence). However, the notion of ending is inherent in the idea of sequential composition: for *this* to happen after *that*, "after" must have some meaning, so *that* must come to an end. By supporting endings as a core part of the abstraction (rather than as something limited to special-purpose objects like buffers or grid sequencers), compositions can be composed at any level. That is, given compositions A and B (which may represent two musical notes, two phrases, two sections, or two movements), one can always concatenate them to produce a new composition. In Aleatora, streams may be sequentially composed with the concatenation operator » (as in `a » b`). The slice operator (as in `a[:10.0]`) is also important, as it enables the composer to convert an infinite stream into a finite one.

Since streams are an augmentation of signals, they inherit support for parallel and functional composition. Streams may be composed in parallel by most arithmetic operators, such as `+` (sum two streams element-wise; audio mixing) and `*` (multiply two streams element-wise; amplitude modulation/enveloping). Parallel composition always implies traversing two streams at the same time, combining them element-wise.[8]

Streams support functional composition most generally via function call. The user can write a function that takes a stream as an argument and produces a new stream. The new stream may advance the given stream as needed to generate its own output, effectively transforming the given stream into a new stream. For example, `fm_osc()` takes a stream of frequencies as an argument. Each time the next sample is requested from the outer stream, `fm_osc` advances the inner stream to obtain the next frequency and updates its phase—transforming a stream of frequencies into a stream of frequency modulation. Aside from this general method, Aleatora also provides `.map()`, `.filter()`, and `.scan()` methods for functional composition, corresponding to the common higher-order functions in functional programming.

[7] A Return object can hold a value, which is important for implementing certain operations such as a lazy split. In terms of `streaming`, this is essential for implementing the monadic bind `(>>=)`.

[8] The fundamental method of parallel composition is `Stream.zip()`, which produces a stream of tuples, with one element from each stream. The other parallel operators are typically built on top of this, and technically represent parallel composition followed by functional composition.

2.3 Implementation

Aleatora is implemented as a library for Python. We chose to implement it as a library so that users can benefit from their existing knowledge of a general-purpose language and from the rich ecosystem of the language. This choice also pairs well with Aleatora's lack of separation between control and synthesis (which allows the composer to work at any level of abstraction without switching languages), as it allows the composer to also work with the entirety of the system (audio, video, files, networking, threading, etc.) without switching languages. Also, this choice saved the authors the trouble of re-implementing (and documenting) an entire standard library's worth of modules which are not directly related to the problem of interest but may nonetheless be required by users.

We chose Python in particular because it is a popular, high-level, dynamic, and "impure" (side-effects can happen anywhere), all of which are desirable for our use case: the composer is typically more concerned with quickly trying out ideas and integrating disparate sources than carefully tracking mutation and I/O in a large system.[9] Conveniently, it also has operator overloading (allowing us to repurpose operators like », +, and slicing), an excellent and mature ecosystem, and a fast JIT implementation (PyPy).

2.4 Usage

Some examples may help make this more concrete. This section presents brief, annotated examples of composition using Aleatora, focusing on core features and stream manipulations. (Examples with networking are presented in the next section.)

```
from core import *  # preliminary imports
from audio import *
# Make an endless stream generating a 440 Hz tone:
tone = osc(440)
# Like `tone`, but ending after the first second:
short_tone = tone[:1.0]
# Play a stream of samples through an audio interface:
play(short_tone)
# Parallel composition:
# Sum streams:
power_chord = (osc(440) + osc(660) + osc(880))/3
# Multiply streams:
ring_mod = osc(440) * osc(660)
amp_mod = osc(440) * (1 + osc(660))/2
# Sequential composition; concatenate streams:
tune = osc(440)[:1.0] >> osc(660)[:1.0]
# `tune` has a pop in the middle from the discontinuity.
# For a smooth transition, we can instead use fm_osc:
tune = fm_osc(const(440)[:1.0] >> const(660)[:1.0])
# This is an example of functional composition;
# the argument to fm_osc is a stream of frequencies,
# which fm_osc transforms into a stream of samples.
# `rand` is an endless stream of random values from 0 to 1.
play(rand*2-1)  # scale to range -1 to 1, play white noise
# We can use it to make stream of pitches:
pitches = (60+(rand*12)).map(int)
```

[9] JavaScript was the runner-up; we may return to it (or one of its derivatives) in the future when we consider the distribution of computational compositions.

```
freqs = pitches.map(m2f)  # or frequencies
chromatic = fm_osc(freqs.hold(1.0))
# We can convert data to streams with to_stream:
list_stream = to_stream([1, 2, 3])
# Calling a stream forces the next value:
list_stream()  # evaluates to (1, <ListStream>)
list_stream()[1]()       # -> (2, <ListStream>)
import wav  # another Aleatora module
# Load audio clips as streams:
a = to_stream(wav.load("sample_a.wav"))
b = to_stream(wav.load("sample_b.wav"))
# Splice b into the middle of a:
spliced = a[:1.0].bind(lambda rest_of_a: b >> rest_of_a)
# Advance `b` at a variable rate, as in varispeed:
wobbly = resample(b, 1+0.3*osc(1))
# Construct a stream that will randomly resolve to
# stream `a` or stream `b` each time it is played:
chance = flip(a, b)
# Make it into an infinite stream, repeatedly choosing:
chances = chance.cycle()
# Live external sources: audio, MIDI
# (Network sources are covered in the next section.)
# Play a clip, then live input (10 seconds), then a clip.
play(a >> input_stream[:10.0] >> b)
# Play live input with a drone forever:
play(input_stream + osc(40))
import midi  # another module
# Play live MIDI input via a sine wave instrument.
play(midi.mono_instrument(midi.event_stream()))
```

All of the discussion so far has been about streams of samples (or numeric data at audio rates), with the assumption that the computer will ultimately "play" the piece (possibly with external input from audience members or performers). However, streams may yield any data type, and the composer can write their piece to "render" down to whatever representation makes sense. For example, one could use Aleatora to compose a piece that generates a stream of MIDI events, which are then rendered as notation for live performance by musicians, as in the pieces described in [6].

2.5 Networking

The stream abstraction is well-suited to data sources such as files, devices, and network connections, which are often exposed by operating systems as streams of bytes or sequences of packets/events. In this section, we describe the networking module, which exposes various kind of network streams as Aleatora streams and facilitates their integration into compositions.

2.5.1 Blocking and Non-Blocking Streams. One issue is that such streams often "block": they may contain large time gaps between subsequent elements. For instance, an OSC message might come in, and another one second later. In between, 44100 samples elapsed where the second element in the stream was not yet ready; if there was a direct dependence between the final output stream and the message stream, this would cause a severe underrun. The same could happen with other network streams, files, or inter-process pipes: all may block due to I/O.

One way to deal with this is to switch to a non-blocking API, which can return a special value or throw an exception if the next value is not ready.[10] However, as this is not always feasible or convenient, Aleatora provides a general mechanism for converting blocking streams to nonblocking streams. The function `net.unblock()` converts a blocking stream into a nonblocking one by running the blocking stream in a separate thread. If values are pulled from the nonblocking stream before new values in the underlying blocking stream are ready, `net.unblock()` can return a "filler" value or repeat the last valid value.

2.5.2 Levels of Networking. Much as Aleatora allows the composer to work at different levels of abstractions in audio (streams may represent samples, MIDI events, etc.), the networking module allows the composer to work at different levels in the network stack.

For example, `net.byte_stream()` connects to a TCP server and returns a stream of bytes, optionally in chunks. Its counterpart, `net.packet_stream()`, returns a stream of UDP datagrams. Slightly higher level, `net.osc_stream()` returns a stream of OSC messages (received over UDP). At a higher level still, it is straightforward to use standard library modules (e.g. urllib) or third-party modules (e.g. Twitter, Wikipedia, Freesound, etc.) with Aleatora, for example via `Stream.map` or `repeat`, which produce streams that repeatedly call a given function.

It's worth describing how all this module fits in to the rest of Aleatora. Generally, network streams mesh well with Aleatora's streams. For example, `net.byte_stream()` naturally ends when there are no more bytes to be read, because the other side closed the connection. This makes it trivial to move ahead in the composition when this happens—automatically and implicitly replacing the defunct parts of the audio graph that used data from the connection. This goes both ways: the UDP and OSC streams are infinite by default (because these protocols are not connection-oriented and have no definite endpoints), but they can easily be ended from above. `net.packet_stream()[:10]` will only listen for the first 10 packets received. And `fm_osc(net.unblock(net.osc_stream().map(lambda p: p.args[0])))[:60.0]` will use OSC messages to control a frequency, but only for one minute; after that, the outer stream ends, stops pulling from the inner stream, and thus stops listening for messages.

This may not be enough to "compose the network" if it were only one-way. But, in addition to being influenced by external streams, Aleatora streams can also have side-effects that may be used to compose the behavior of other devices on a network. Examples of this usage, as well as the functions described earlier, are given in the annotated listing below.

```
from core import *              # Aleatora module
from audio import *            # Aleatora module

## Examples with third-party libraries.
from speech import speech      # Aleatora module
import wikipedia               # 3rd-party module
# This stream speaks endless Wikipedia article titles.
wiki = repeat(wikipedia.random).map(speech).join()
# We can control this at either level: text or samples.
```

[10] For example, using `MSG_DONTWAIT` with `recv()` for Linux sockets.

```
# For example, this stops after ten titles:
wiki = repeat(wikipedia.random)[:10].map(speech).join()
# whereas this stops after ten samples:
wiki = repeat(wikipedia.random).map(speech).join()[:10]
# This reverses the titles before saying them (as text):
wiki = (repeat(wikipedia.random)
        .map(lambda t: t[::-1])
        .map(speech)
        .join())
# while this reverses titles after saying them (as audio):
wiki = (repeat(wikipedia.random)
        .map(speech)
        .map(Stream.reverse)
        .join())

import net                              # Aleatora
from cryptocompare import get_price  # 3rd-party
# Stream of Ethereum prices in USD.
b = repeat(lambda: get_price('ETH','USD')['ETH']['USD'])
# Stream of ether prices as frequencies.
# (Around 2700 Hz at the time of writing.)
e = fm_osc(net.unblock(b, filler=0, hold=True))

## Examples with OSC.
# Stream of frequencies via OSC over UDP (port 8000):
freqs = (net.osc_stream()
    .filter(lambda m: m.address == b'/freq')
    .map(lambda m: m.args[0]))
held_freqs = net.unblock(freqs, filler=0, hold=True)
# Stream of frequencies, externally controlled by OSC:
controlled = fm_osc(held_freqs)
# Define a new stream function:
def switch(a, b, dur):
    # Switch between streams a and b at regular intervals.
    return a[:dur].bind(lambda rest: switch(b, rest, dur))
# Semi-controlled; switch from controlled to random freqs.
rand_freqs = (rand*1000).map(int)
semi = fm_osc(switch(held_freqs, rand_freqs, 1.0))

## Examples with side-effects
#  (The composition controls devices on the network.)
# For each random frequency generated,
# send it to another device over OSC.
from oscpy.client import OSCClient  # 3rd-party
client = OSCClient('10.0.0.2', 8000)
effectful_freqs = rand_freqs.each(
    lambda f: client.send_message(b'/freq', [f]))
# Generate, play, and send out frequencies once per second.
play(fm_osc(effectful_freqs.hold(1.0)))
# If we don't want to generate audio locally,
# (that is, if we *just* want the side-effects)
# we can iterate through streams directly:
import time  # standard library module
for _ in effectful_freqs: time.sleep(1)

# Every time live audio input exceeds a given level, send
# an event. The stream is silenced by the map at the end.
def warn(x):
    if abs(x) > 0.8:
        client.send_message(b'/too_hot', [x])
    return x

play(input_stream.map(warn))
```

Hopefully, these examples provide some insight into Aleatora, effectful streams, and how they can be used to compose the network and mix external control with internal structure. Looking ahead, we plan to build out more networking support, particularly for compositions involving many simultaneous network streams—for example, a system for providing one stream for each active WebSocket connection in an audience participation piece.

3 CONCLUSION AND FUTURE WORK

To paraphrase Alan Kay, Aleatora aims to make easy things easy and hard things possible. We believe Aleatora's streams are a good abstraction for this goal: all three fundamental forms of composition are supported by the core abstraction. The composer is empowered to work at whatever level of abstraction they need to. Whether they are working in the world of notes and events, in the world of patterns and phrases, or all the way down in the world of samples and signals, the framework will support them, switching between layers with as little pain as possible. Likewise, whether they are working with static or dynamic media, data sources, or compositions, the framework will support them, enabling them to mix and match and manipulate everything with a consistent interface.

Looking ahead, we hope to evaluate Aleatora with composers and livecoders, understand how may be used in practice, and consider how it might fit into a more complete compositional environment. We also intend to refine Aleatora's implementation, improve performance, and evaluate the feasibility of running computational compositions written with Aleatora in the browser.

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman. 1996. 3.5 Streams. In *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts. https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html

[2] Philip L. Bewig. 2007. SRFI-41: Streams. https://srfi.schemers.org/srfi-41/srfi-41.html

[3] Roger B. Dannenberg. 1997. The Implementation of Nyquist, A Sound Synthesis Language. *Computer Music Journal* 21, 3 (1997), 71–82. https://doi.org/10.2307/3681015 Publisher: The MIT Press.

[4] Roger B. Dannenberg. 1997. Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. *Computer Music Journal* 21, 3 (1997), 50–60. https://doi.org/10.2307/3681013 Publisher: The MIT Press.

[5] Karlheinz Essl. 2007. *Algorithmic composition*. Cambridge University Press, 107–125. https://doi.org/10.1017/CCOL9780521868617.008

[6] Jason Freeman. 2008. Extreme Sight-Reading, Mediated Expression, and Audience Participation: Real-Time Music Notation in Live Performance. *Computer Music Journal* 32, 3 (2008), 25–41. http://www.jstor.org/stable/40072645

[7] L. Turchet. 2019. Smart Musical Instruments: Vision, Design Principles, and Future Directions. *IEEE Access* 7 (2019), 8944–8963. https://doi.org/10.1109/ACCESS.2018.2876891

[8] Luca Turchet, Michele Benincaso, and Carlo Fischione. 2017. Examples of Use Cases with Smart Instruments. In *Proceedings of the 12th International Audio Mostly Conference on Augmented and Participatory Sound and Music Experiences* (London, United Kingdom) *(AM '17)*. Association for Computing Machinery, New York, NY, USA, Article 47, 5 pages. https://doi.org/10.1145/3123514.3123553

[9] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet. 2018. Internet of Musical Things: Vision and Challenges. *IEEE Access* 6 (2018), 61994–62017. https://doi.org/10.1109/ACCESS.2018.2872625