Composing the Network with Streams

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

Jason Freeman
jason.freeman@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 facilitate 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 and Jason Freeman. 2021. Composing the Network with Streams. In *Audio Mostly 2021 (AM '21), September 1–3, 2021, virtual/Trento, Italy.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3478384.3478416

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.

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 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 weave 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 [10] and Internet of Musical Things [11]) has focused on the instrument: Internet-enabled *smart instruments* that combine sensors, computational capabilities, and network connectivity [8]. For example, [9] 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 [11] — 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 and the operations they support.[2] 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 core features, including an interactive environment based on a dynamic language with a REPL,[3] a lack of distinction between score and orchestra, sounds as first-class values, streams ("signals" in Nyquist) as potentially-infinite lazy linked-lists, and support for arbitrarily mixed sample rates.

AM '21, September 1–3, 2021, virtual/Trento, Italy
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8569-5/21/09…$15.00
https://doi.org/10.1145/3478384.3478416

[1]e.g. Max/MSP, Pure Data, SuperCollider, Csound, ChucK, JavaScript with Web Audio
[2]SuperCollider also has streams for lazy, potentially-infinite sequences, but they are only used for control (not synthesis) and cannot be replayed from arbitrary points. (The latter point also distinguishes Aleatora streams from Python generators.)
[3]That is, a read-eval-print-loop, or interactive language shell

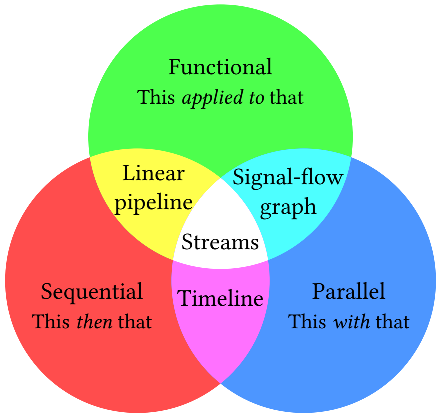

Figure 1: Types of composition and the abstractions that support them.

That said, our work differs from Nyquist [3] in a few essential ways. 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 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, 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])

In 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 [1] and [2], 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 Figure 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.

[4]Although the latter can be attained in part through operator/method overloading.
[5]https://hackage.haskell.org/package/streaming

Sequential composition entails concatenating two processes, 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 `;` or newline operator. Musically, sequential composition corresponds to playing one thing after another, splicing tape, placing one clip to the right of another in a DAW, or writing one note after another in Western music notation.

Parallel composition entails running two processes at the same time.[6] 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. Musically, parallel composition corresponds to playing two things at the same time, placing one clip on top of another (on another track) in the DAW, 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 with another process. This is akin to linking two guitar pedals via 1/4" cable, connecting objects in Pd or MSP, calling `AudioNode.connect()` in the WebAudio API, or using the => operator on UGens in ChucK. In Western music, this is represented by notation that affects a performer's interpretation (dynamics, instructions like "pizzicato," "vibrato," etc.) or 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 is the essence of the signal flow graph. It is well-supported by audio programming languages and is a natural feature of circuits (and thus is the core of analog and modular synthesis). Sequential/parallel composition is the essence of the timeline (with horizontal time and vertical layers), and it is well-supported in multitrack tape and the DAW. Sequential/functional composition is the essence of the linear pipeline (the operators `;` and `|` in Unix shells, as in CARL [7]).

This correspondence also implies that each of these 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 sequential composition, so most audio programming languages augment it with an imperative language that can manipulate the graph. The imperative code can be sequenced, but this does not concatenate the output; the execution of the audio graph is mostly 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 dwindle

[6]Note that this does not require that the processes run in parallel (hardware parallelism), just that both can contribute to the same samples in the output.
[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 (`>>=`).

to zero (silence). However, the notion of ending is inherent in the idea of sequential composition: for *that* to happen after *this*, "after" must have some meaning, so *this* must come to an end. By supporting endings as 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 via binary operators such as + (sum two streams element-wise; audio mixing) and * (multiply two streams element-wise; amplitude modulation/enveloping). Parallel composition always implies traversing multiple streams together. [8]

Streams support functional composition 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. 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, higher-order functions common in functional programming.

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, 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. 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).

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

2.4 Usage

This section presents brief, annotated examples in Aleatora (with import statements omitted), focusing on core features and stream manipulations. Examples with networking are presented in the next section. The results can be heard at youtu.be/MJwjYW1jEc0.

```
tone = osc(440)  # Endless stream: 440 Hz tone
short_tone = tone[:1.0]  # End after the first second
play(short_tone)  # Play stream of samples live
# Parallel composition:
power_chord = (osc(440)+osc(660)+osc(880))/3  # sum
amp_mod = osc(440) * (1 + osc(660))/2   # multiply
# Sequential composition:
tune = osc(440)[:1.0] >> osc(660)[:1.0] # slice, concat
# Functional composition:
# `tune` has a pop in the middle from the discontinuity.
# For a smooth transition, we can instead use `fm_osc`,
# which turns a stream of frequencies into one of samples:
tune = fm_osc(const(440)[:1.0] >> const(660)[:1.0])
# `rand` is an endless stream of random values from 0 to 1
# We can use it to make a stream of pitches:
pitches = (rand*12+60).map(int)
freqs = pitches.map(m2f)  # then frequencies
chromatic = fm_osc(freqs.hold(1.0))  # then samples
# 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))
# Randomly resolves to stream `a` or `b` each play:
chance = flip(a, b)
# Make it into an infinite stream, repeatedly choosing:
chances = chance.cycle()
# Play live audio input, plus a drone, forever:
play(input_stream + osc(40))
# Play a clip, then live input (10 seconds), then a clip.
play(a >> input_stream[:10.0] >> b)
# Play live MIDI input via a sine wave instrument.
play(midi.mono_instrument(midi.input_stream()))
```

The discussion so far has mostly been about streams of samples, 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 generate whatever representation makes sense. For example, one could use Aleatora to compose a piece that yields a stream of MIDI events, which are then rendered as notation for live performers, as in the pieces described in [6].

2.5 Networking

Streams are well-suited to sources such as files, devices, and network connections, which are exposed by operating systems as streams of bytes or sequences of packets/events. In this section, we describe the networking module, which exposes 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": it may take a long time to compute the next element as the system waits for the next packet to come in. A direct dependence between a blocking I/O stream and the output stream will cause buffer underruns. Some APIs offer non-blocking options;[9] as these are not always available or convenient, Aleatora offers a general mechanism, `unblock()` for converting a stream from blocking to nonblocking by running it in a separate thread. If values are pulled from the nonblocking stream before new values in the underlying blocking stream are ready, `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, `byte_stream()` yields a stream of bytes over a TCP connection, and `packet_stream()` yields a stream of incoming UDP datagrams; at a higher level, `osc_stream()` returns a stream of OSC messages; beyond that, it is easy to use other modules (e.g. urllib, or third-party modules for Twitter, Wikipedia, Freesound, etc.) with Aleatora, as will demonstrated shortly.

Network streams mesh well with Aleatora. `byte_stream()` ends when there are no more bytes in the TCP stream, so it's trivial to move ahead in a composition when the connection closes—automatically replacing the defunct parts of the audio graph that relied on the connection. This goes both ways: the UDP & OSC streams are infinite (as the protocols are not connection-oriented) but can easily be ended from above (e.g. to stop after a particular message or amount of time).

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 (Aleatora imports omitted for brevity):

```
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 audio.
# 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())
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 played as frequencies:
e = fm_osc(net.unblock(b, filler=0, hold=True))
# 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 stream function: switch between a and b regularly.
def switch(a, b, dur):
    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))
# For each frequency generated, send it to a device over OSC.
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)))
```

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

3 FUTURE WORK

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

The source code for the version of Aleatora described in this paper is available at github.com/ijc8/aleatora/tree/am21. The latest version is available at github.com/ijc8/aleatora.

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. In *The Cambridge Companion to Electronic Music*, Nick Collins and Julio d'Escrivan (Eds.). 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] Gareth Loy. 2002. The CARL System: Premises, History, and Fate. *Computer Music Journal* 26, 4 (2002), 52–60. http://www.jstor.org/stable/3681769

[8] 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

[9] 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

[10] Luca Turchet, György Fazekas, Mathieu Lagrange, Hossein S. Ghadikolaei, and Carlo Fischione. 2020. The Internet of Audio Things: State of the Art, Vision, and Challenges. *IEEE Internet of Things Journal* 7, 10 (2020), 10233–10249. https://doi.org/10.1109/JIOT.2020.2997047

[11] 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