Vesa Norilo and Roger B. Dannenberg, "KO2 - Distributed Music Systems with O2 and Kronos," in *Proceedings of the 15th Sound and Music Computing Conference (SMC2018),* Cyprus, 2018, pp. 452-456.

KO2 - DISTRIBUTED MUSIC SYSTEMS WITH O2 AND KRONOS

Vesa Norilo
University of the Arts Helsinki
vno11100@uniarts.fi

Roger B. Dannenberg
Carnegie Mellon University
rbd@cs.cmu.edu

ABSTRACT

KO2 is a platform for distributed musical applications, consisting of the messaging protocol O2 and the signal processing language Kronos. This study is an effort to use O2 as a comprehensive communications framework for inter-process signal routing, including clock synchronization and audio. The Kronos compiler is exposed as an O2 service, allowing remotely specified programs to be compiled and run in near real-time on various devices in the network.

1. INTRODUCTION

With computing devices becoming lightweight and ubiquitous, artworks are being built from an increasingly wide array of devices. Software systems are distributed across multiple devices, and the need for robust and coherent communication arises.

Some of the technological challenges involved in distributed music systems include communication and deployment. Software must be pushed to all the involved devices, initialized, and coordinated during run-time. In addition, audio signals are routed throughout, often with a separate signal stack.

Several communication protocols exist with the aim of allowing independent musical processes to communicate. In this paper we use one of them, O2, to develop a framework for distributed processing. We study the feasibility of using O2 for audio distribution as well.

A novel addition to the field is the inclusion of a programming language, Kronos, in the system. With Just-in-Time compilation, distributed systems can contain compute server nodes that compile and execute remotely specified programs in near real-time.

The rest of this paper is organized as follows: In Section 2, *Background*, we examine the component systems O2 and Kronos and describe some recent additions to them. In addition, prior art related to distributed systems is surveyed. Section 3, *Implementation*, details the design choices behind the framework. Usage and performance is discussed in Section 4, *Discussion*, followed by possible *Future Work* and *Conclusion* in Sections 5 and 6.

2. BACKGROUND

O2 can be seen as an extension of OSC. [1] It uses URL-like address strings and patterns to name the destinations of messages containing a list of typed values. OSC assumes nothing about the transport and typically requires users to enter IP addresses of servers. In contrast, O2 has a built-in discovery mechanism to locate named services automatically, and so addresses always begin with a service name. In addition, O2 messages are all timestamped, and O2 performs clock synchronization so that timestamps are meaningful when messages cross from one host to another. Finally, O2 offers a choice of reliable (TCP) or best-effort (UDP) message delivery.

Kronos is a functional audio programming language, an idea pioneered by the FAUST project. [2][3] It aims to provide a powerful, expressive language for describing fixed-graph signal flows in a multi-rate signal processing environment. Kronos was designed to operate on the abstractive level of unit generators and orchestras. Recent additions explore the scheduling of musical events and score-level constructs. [4]

2.1 Recent Developments

2.1.1 Kronos Meta-Sequencer

Recently, Kronos was enhanced to support dynamic instantiation of signal processor objects as well as scheduling control messages or arbitrary code by integrating a custom interpreter with a sequencer and the JiT compiler. [4]

One of the goals of the Kronos compiler system is to enable deployment of stand-alone applications. The meta-sequencer implementation was not readily adaptable to this scenario, because of its coupling with the compiler.

Recently, the meta-sequencer was refactored to remove the interpreter entirely. Its functionality was moved to a small runtime library that Kronos programs can call directly via the foreign function interface. As a result, stand-alone applications can now use dynamic instantiation and scheduling by linking against the runtime library, without needing the complete runtime environment or the compiler.

2.1.2 O2 Hub

O2's original discovery mechanism works well in local area networks that allow UDP broadcast, but it was found that many organizations block WiFi broadcast messages. An extension to O2 allows discovery to be managed by any O2 process designated as a "hub." To use a hub, processes must obtain the IP address of the hub manually or through some external protocol. Processes can then use the hub to

discover all other O2 processes. The hub idea works equally well over wide area networks.
O2 has also been extended with connections called "taps," which forward incoming messages from a tapped service to some other service, supporting debugging and remote monitoring.

2.2 Distributed Music Systems

2.2.1 Audio over the network
Various systems have explored audio transmission over shared local and wide area networks using IP (Internet Protocol) as opposed to specialized real-time protocols and dedicated hardware. The Dante system[1] sends audio over IP using hardware support and appears to applications as an audio I/O device. Ravenna is an open standard for media over IP using the standard real-time streaming protocol (RTSP). [5] JackTrip is an application that sends audio streams over UDP and has been widely used for live performance over long distances. [6] AuraRT explored distributed computation for audio synthesis and control in a local area network. [7] Our work with O2 is most similar to AuraRT in that we are basing audio streaming on a general message-passing system that integrates clock synchronization, global object naming and timed messages between distributed objects.

2.2.2 Remote Just-in-Time Compilation
The Java platform is a significant example of remote compilation. The Java Remote Method Invocation enables a virtual machine to download and execute bytecode from the network. RMI is used in Apache Hadoop, a distributed storage system, to run custom search queries over multiple storage nodes.[2]

2.2.3 Service and Namespace Discovery
OpenSoundWorld is notable in that it implemented a query protocol for OSC where a client can query a server for information about the address space and the parameters expected by each addressable node. [8]

O2 supports multiple *services,* which are abstractions of servers that process O2 messages. Any O2 *process* can create one or more services, and services are automatically discovered by other processes. Message addresses begin with a service name, and O2 routes messages to the corresponding *service*, which may be in the same process, on the same host computer, or on a remote computer.

3. IMPLEMENTATION

This section describes the various components that constitute the KO2 system: the O2 Audio protocol, O2 Namespace Discovery and the Kronos Compile Server.

3.1 Object Models in O2 and Kronos

Object oriented designs are typical in music systems. O2 services are instances of objects within an application, where "application" in O2 refers to a collection of cooperating processes that may be distributed across a number of networked machines. O2 messages correspond to method calls that have no return value.

3.1.1 Data Types in O2
As O2 is concerned with serialization of messages for transport over the wire, datatypes play a vital role. The O2 model is a straightforward extension of the OSC [1] system of type string accompanied by binary data.

However, in OSC, methods are nothing but addresses, and no provisions are made to associate type information with methods; it is up to each individual method to parse the parameter types.

In O2, message handlers (i.e. methods) may be associated with type strings of their own. Any method can be type-coerced, so that exact type matches are not required, yet it still works as expected. This is similar to dynamically typed languages in the sense that the data types on the wire are an implementation detail of the run-time system.

3.1.2 Kronos Signal Processor as an Object
Kronos models signal processors as discrete reactive systems – sequences of output events as a function of sequences of input events. [9] The fundamental properties of an object are state, identity and behavior. At the language level, Kronos provides no concept of state or identity. However, state is used internally by the compiler.

Mattheussen has demonstrated automated translation between the traditional object model and Faust [2], a functional audio language. [10] Similarly, Kronos signal processors appear as objects to the user, particularly with the dynamic instantiation extension. As previously described, the reactive inputs of a Kronos signal processor closely resemble the methods of an object.

Passing O2 messages to Kronos objects is a natural fit; the O2 methods and their type strings can be automatically derived from the Kronos program and type coercion enabled if desired.

3.2 O2 Namespace Discovery

In an environment where services can be built dynamically, it is important to provide a mechanism for self-reflection: enable components of the distributed system to query each other for available methods.

Like audio, namespace discovery can be built as a user-space protocol on top of the O2 core. We implemented this as a directory service that runs as a process within any O2 application. Conforming services can register or de-register their methods with the directory service.

3.2.1 Reply messages
The directory service supports the querying of O2 namespaces for method addresses, type strings and documentation. In addition, methods can be searched for with regular expressions. The directory service is detailed in Table 1.

All the query messages require data to be returned to the sender, for which there is no built-in facility in O2. We

[1] https://www.audinate.com/node/128

[2] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

adopted a reply convention that is already used internally in the O2 clock synchronization protocol.

Each query message contains a 64-bit message identifier, which the sender may assign arbitrarily, and a reply address. The reply address is appended with `/get-reply`, and the message identifier is sent along with the query results.

Method name	Type	Description
`add-method`	`Sss`	*Registers a method by address pattern, type string and documentation*
`remove-service`	`sS`	*Removes all registrations in a namespace*
`Regex`	`hss` `Hss`	*Retrieves all method metadata where the address pattern matches the supplied regular expression. The first two parameters are query id and reply address.*

Table 1. Methods implemented by the directory service

3.3 Audio over O2

Rather than extend the O2 core library with audio-specific functionality, we decided to implement network audio as a protocol in the O2 user space. This avoids further complicating the core of O2.

O2 Audio is intended as a proof of concept and not expected to outperform the state of art, but we hope that it proves useful for distributed audio applications, especially as the existing infrastructure for clock synchronization and service discovery can be used.

3.3.1 Design

O2 Audio involves sending buffers of audio samples as messages over the reliable TCP transport, which guarantees in-order delivery. Receiving audio endpoints are O2 methods, with globally unique addresses in the form of `/service/endpoint`. The audio is passed as vectors of single-precision floating point values, for convenience and simplicity at the cost of higher network bandwidth utilization.

Each endpoint provides a summing mixer, and any process within the O2 application can push audio to it. O2 audio can also be synchronized.

The audio stream can be seen as a contiguous vector of samples the receiver reads from. Each stream has a globally unique identifier, and the summing mixer maintains the location of the write head for it.

A sender can reposition its write head with a synchronization message. Because O2 provides global clock synchronization, multiple senders can produce audio streams that are guaranteed to be synchronously read by the receiver, even if the data is delivered ahead of time to account for network and playback latencies.

3.3.2 Implementation

The summing mixer is backed by a ring buffer that represents a view into the audio stream, based on the location of the read head. Incoming audio buffers are only consumed if they are within the time range currently in the view. Incoming buffers with timestamps falling behind the read head indicate that the latency of the network is too big for the requested timing.

Buffers too far ahead of the read head are timestamped and scheduled by O2 and delivered at the appropriate time. For a detailed description of the O2 audio protocol methods, please refer to Table 2.

For now, packet ordering is relegated to the protocol layer. The system is only useful on well-performing, non-congested networks without packet loss. Many UDP-based protocols include packet loss recovery and custom ordering logic. However, O2 supports using a mix of UDP and TCP messages, so reliable transmission can still be achieved over congested networks if more latency is acceptable.

Method name	Type	Description
`<endpoint>/push`	`Hvf`	*Sum vector of floating point samples by stream identifier*
`<endpoint>/sync`	`Ht`	*Position stream by identifier according to the time point*
`<endpoint>/close`	`H`	*Dispose the stream and expect no further samples*

Table 2. Methods implemented by the audio receiver endpoint service

3.4 Kronos JiT Compile Server as a O2 Service

Dynamic creation of new services is facilitated by the Kronos compiler, itself exposed as a O2 service: its main method compiles textual source code into an executable signal processor and wraps it in a new O2 service, as described in Section 3.1.2. The compiler methods are shown in Table 3.

Each compile server node provides a mix bus for all the signal processor instances within. Depending on the configuration, the mix bus is either played back on local audio hardware or pushed to an O2 Audio endpoint.

3.4.1 Example

A simple synthesizer written in the Kronos language is shown in Listing 1. One could invoke the compile server with **`/new foo <source-code>`**, and the resulting instance would then expose the O2 methods **`/foo/vibr-freq`**, **`/foo/vibr-depth`** and **`/foo/freq`**.

```
vf = Control:Param("vibr-freq" 6)
vd = Control:Param("vibr-depth" 0.05)
freq = Control:Param("freq" 440)

Wave:Sin(freq * 1 + Wave:Sin(vf) * vd)
```

Listing 1. Simple synthesizer written in the Kronos language.

Method name	Type	Description
`/new`	`ss`	*Compile the received string (2) as Kronos code into an instance, binding it to the received name string (1)*
`/delete`	`s`	*Destruct the instance bound to the name string*

Table 3. Methods implemented by the JiT compile server

4. DISCUSSION

So far, we have presented the technical foundation of the KO2 system. In this section, we envision usage scenarios, as well as quantify and measure the performance of the system.

4.1 Collaborative Live Coding

In this scenario, several performers write code on stage. There is a centralized compile and compute server, and the performers' machines are thin clients. Each client can push program code to the server via O2 messages: the server compiles and runs the code and outputs audio.

What makes this setup more interesting that simply having several independent machines is the fact that the performers could share code, due to having a common a JiT context. Namespace discovery allows the performers to be informed about the capabilities of new processes as they appear in the system.

We have not yet tried this scenario in practice. It is probable that sharing code increases the risk of breakage during performance. The impact and mitigation, as well as the relative benefit remains to be investigated.

4.2 Distributed Audio Processing

Inverting the previously outlined scenario yields a centralized controller with several compile servers. This setup could be used in an installation, where signal processors need to be in the proximity of sensors or actuators.

Notably, programs can be quickly deployed to the compute nodes without restarting or reconfiguring them. The compile server can run on low-power devices, as long as the architecture is one of the many supported by LLVM, and a full operating system, such as GNU/Linux, is present.

4.3 O2 Audio Latency and Performance

Merging Technologies Inc. have implemented a commercial system around network audio, based on the Ravenna protocol. They claim reliable operation with a throughput latency of 1.5 milliseconds, in a networked system consisting of an analog-digital-analog converter connected to a Windows PC with a real-time hypervisor and a powerful network interface. We have not independently verified this number but consider it to be an indication of the upper bound in performance attainable with IP-based protocols.

4.3.1 Bandwidth Requirement

There are several layers of overhead, some due to the O2 Audio protocol itself, while some are intrinsic to the TCP/IP transport.

O2 Audio uses 32-bit floating point samples, which nearly doubles the bandwidth requirement. However, it should have very little impact on latency, the parameter we consider the most critical in network audio.

Some additional overhead is caused by the O2 Audio protocol, which encodes endpoint address, type string and a stream identifier for each transmitted buffer. For the numbers given below, we assume an address pattern of 12 characters, that an audio buffer of 256 samples is sent in a single packet, and that the common IP MSS of 1460 bytes is used. In this case, 94% of the transmitted bits are used for audio data. For details, please refer to Table 4.

Label	**Bytes**
O2 header	20
Sample data	1024
TCP/IP header	40
Total	**1084**
Overhead	**5.86%**
Bandwidth for 44.1kHz audio	**1459 kbps**

Table 4. Summary of O2 Audio bandwidth for a buffer size of 256 samples and 44.1kHz sample rate

For a 54 Mbps 802.11g network link, the theoretical maximum number of transmissible audio channels at 44.1kHz sample rate is 37, while a 1Gbps link might be able to support over 700 channels.

4.3.2 Latency Measurements

We measured the round-trip latency of O2 Audio between two processes. One of them is a simple loopback service that performs the computationally trivial operation of inverting the signal polarity. The master process generates an audio stream and sends it for processing in 1000 sample buffers, and measures the time until the response buffer has been received.

The measurement was performed using `std::high_resolution_clock` in C++ under three different circumstances: two O2 processes coexisting in a single operating system process, each O2 process as a distinct operating system process, and finally, each process running on a different computer in a WiFi network. For the single machine measurement we used a Windows 10 PC with a 2.4GHz Core i5-6300U processor. For the distributed measurement we added a MacOS computer with a 1.8GHz Core i5 processor. The WiFi network was set up with the Windows machine hosting a network over the ASUS AC51 WiFi adapter.

Table 5 details the measurements; for each scenario, we report the median roundtrip latency, as well as the latency value that was higher than 90% of the measured values. The latter is more indicative of the worst-case performance which is relevant in the use case of stable low latency audio. Interestingly, the interprocess latency is slightly lower than the in-process latency. This may be due to a higher level of resource contention in the case of two aggressively polling threads sharing an O2 instance. The interprocess scenario may be helped by the loopback socket optimizations in Windows 10.

Scenario	Median	90th percentile
In-process	0.49ms	0.83ms
Interprocess	0.42ms	0.73ms
WiFi	2.45ms	3.16ms

Table 5. O2 Audio latency measurements

5. FUTURE WORK

5.1 Audio over O2

The current audio implementation is fixed to use TCP/IP and single precision floating point buffers. Higher performance might be attained, especially in poor network conditions, with UDP and lower transport resolution.

While the latency figures, as shown in Table 5, are good, it remains to be seen how the latency characteristics hold up under more adverse conditions, such as network load or CPU-bound scenarios.

5.2 Security

Information security is part of any networked computer system. KO2 is wide open: the compile server allows arbitrary remote code execution by design. Malignant actors must be kept out at the network level.
Perhaps some form of access control could be built into KO2, provided it would not make system setup and initialization more difficult.

6. CONCLUSIONS

This study proposes a framework for building distributed music applications, with synchronization, communications and inter-process signal routing provided by O2, and dynamic programming provided by Kronos.

We described the implementation of audio transport over the network via O2 messages, as well as dynamic compilation of program code and automated discovery of new services and methods.

In total, the system provides a rare combination of a comprehensive platform for distributed applications with, to our knowledge, the first remote JiT-compilation capable music language.

The software described in this study is open source and freely available.[3] Contributions and users are welcome.

Acknowledgments

Vesa Norilo's work has been supported by the Academy of Finland, award number SA311535.

7. REFERENCES

[1] M. Wright and A. Freed, "OpenSound Control: A New Protocol for Communicating with Sound Synthesizers," in *Proc. of the 1997 ICMC*, 101-104.

[2] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust." *Soft Computing* 8(9):623-632, 2004.

[3] V. Norilo, "Kronos: A declarative metaprogramming language for digital signal processing." *Computer Music Journal* 39(4):30-48, 2015.

[4] V. Norilo, "Kronos Meta-Sequencer - From Ugens to Orchestra, Score and Beyond." *In Proc. Of the Int. Computer Music Conf.* (ICMC), Utrecht, 2016.

[5] Ravenna, "AES67 and RAVENNA in a Nutshell," available: https://www.ravenna-network.com/resources/, accessed April 6, 2018.

[6] J.-P. Cáceres and C. Chafe, "JackTrip: Under the Hood of an Engine for Network Audio," in *Proc. of the Int. Computer Music Conf.* (ICMC), Montreal, 2009, 509-512.

[7] R. B. Dannenberg and P. van de Lageweg, "A System Supporting Flexible Distributed Real-Time Music Processing," in *Proceedings of the 2001 Int. Computer Music Conf.* (ICMC), San Francisco, 2001, 267-270.

[8] A. Chaudhary, A.Freed and M. Wright, "An Open Architecture for Real-time Music Software," in *Proc. of the 2000 Int. Computer Music Conf.* (ICMC), Berlin, 2000.

[9] Van Roy, Peter. "Programming paradigms for dummies: What every programmer should know." *New computational paradigms for computer music*. IRCAM, 2009.

[10] K. Matheussen, "Poing Impératif: Compiling Imperative and Object Oriented Code to Faust." *In Proc. of the Linux Audio Conf* (LAC), 2011.

[11] V. Lazzarini, "Audio Signal Processing and Object-Oriented Systems.", *in Proc. of the Int. Conf on DAFx*, 2002.

[3] https://github.com/rbdannenberg/o2 https://bitbucket.org/vnorilo/k3