Sphactor: Actor Model Concurrency for Creatives

initial design of a new framework

Arnaud Loonstra <arnaud@sphaero.org>,
Aaron Oostdijk <aaron.oostdijk@hku.nl>,
Mikal van Leeuwen <mikal@dehoofdwerker.nl>
January 20 2020 (draft version)
Expertise Centre Creative Technology, Utrecht University of the Arts

Abstract

We propose a combined visual and text-based programming environment based on the actor model suitable for novice to expert programmers. This model encompasses simple communicating entities which easily scale from utilizing threads inside the computer to massive distributed computer systems. To design our proposed environment we classify different levels of programming users encounter when dealing with technologies in creative scenarios. We use this classification system as a foundation to design our proposed environment to support (novice) users on their way to a next level. This framework not only intends to support modern computing power through a concurrent programming paradigm, but is also intended to let users interact with it on the different classification levels.

Introduction

This research is related to enabling novice users to utilize computer technology. In general we can say that every novice user finds it hard to utilize modern technologies. Especially very contemporary popular technologies such as Mixed Reality require a user to know programming to fully operate the technology. While novice users are very interested in these technologies, they usually do not posses the necessary skills and can find these technologies to be intimidating.

As an example; within the university students want to work with motion capture technologies. However as soon as they want to do custom things with the sensors they need to work with the manufacturer's C++ SDK. This is often beyond their current skill level and it would take a significant amount of time and effort to get them there by themselves. Moreover, it is beyond the current curriculum for these students to get them to the required level of programming proficiency.

To overcome this hurdle we have created a simple software tool which translates the motion capture data to Open Sound Control (OSC)[1]. The reason we translate to OSC was due to the fact that all tools operated by our students can receive OSC out of the box.

By doing this we noticed the barrier for operating the motion capture technology was lowered sufficiently for students to become very creative with the possibilities of capturing motion. As this is was the intended goal of the exercise for our students we began plotting how we could expand this to more technologies.

Extrapolating what would be needed to do this more showed an immediate jungle of tools emerging with a maintenance challenge to keep it running on students machines. Therefore we wanted to develop an environment in which we could connect any number of systems. We envisioned this as an intermediate software layer in which we could plug any technology. Ideally this could plug into other tools instead of being its own tool.

Besides lowering the barriers for newcomers we want to embed a didactic view on utilizing technologies. As we want our students to move from solely ‘using’ to also ‘operating’ and programming technologies, we would like to give them a sustainable learning path. We often see tools that are comfortable for students to use but limit them in the long run when the student wants more than the tool can provide.

Finally we notice that current student’s computers are equipped with multiple processors but the tools they operate sporadically utilize all these processors. We therefore researched how utilizing multiple processors could be integrated from the start.

In this research, we develop a first prototype with the specific intent to address the above mentioned situations while also providing us with intermediate technology we can improve on.

Related Work

Creative coding is a phrase coined by John Maeda in 2004 [2]. We refer to this term when we see artists generating expressions using computer technology through programming.

We talk of a Creative Coding ‘tool’ when it is a software environment in which one can program to generate output. We talk of a Creative Coding ‘framework’ when one can program to create an application which generates output.

There are many tools and frameworks available for Creative Coding. Here we will only mention those relevant for our research or when having a strong presence in Creative Coding practices.

Max[3] and Puredata[4] are most popular Dataflow programming tools which have a strong presence in audio practices. One can program using flows of data from one block to another. While neither tools are able to utilize multiple processors, the Dataflow programming model has similarities to the Actor Model[5], which we will mention further on, because of its sending and receiving of data.

Isadora[6] is a tool which is very welcoming for novice users. It’s an environment in which one can create interactive scenes using actors. It appears that Isadora operates on an Actor Model but it it seems to do so non-concurrently. We cannot confirm this because of its proprietary nature.

Touchdesigner[7] is another Creative Coding tool mostly geared towards visual output. We are only recently seeing Touchdesigner being used by artists. It seems to operate on a Dataflow model which can utilize the GPU or offload to other CPU’s for some operations. It is regarded as more complex than Isadora for example and therefore more intimidating for novice users.

Processing[8] and OpenFrameworks[9] are the most popular Creative Coding frameworks in which one can program in Java or C++ respectively. Both frameworks operate on the same principle of a game application loop[10] consisting of setup(), update() and draw() methods. While Processing is the go-to framework for newcomers, OpenFrameworks seems to better cater for serious programming practices.

OpenFrameworks is very popular creative coding frameworks but because it uses C++ it is regarded as much more intimidating than Processing. Setting up the OpenFrameworks environment requires considerable more effort than simply installing Processing. Understanding OpenFrameworks error messages implies knowledge about compilers and linkers which no novice user will have. Moreover, we have noticed time and time again that text based programming is only picked up by newcomers if they explicitly need to do so.

It is therefore interesting to mention the developments around the Mosaic[11] environment. Mosaic is a Visual Patching tool built on top of the OpenFrameworks eco-system. It is still in development but seems to merge OpenFrameworks with visual programming and higher level programming languages such as Lua and Python.

Another interesting project in this regard is the OSSIA[12] project which is an interactive sequencer and corresponding libraries. It utilizes the OSC protocol as well as a discovery protocol. OSSIA is specially aimed at using OSC to control other tools and uses as specially designed UI to create interactive sequences.

We notice students own computers equipped with multiple processors. However the tools they operate, like the tools we mentioned, sporadically utilize all processors inside the computer.

Since around 2003 CPU manufacturers are unable to manufacture faster CPU's anymore[13]. Instead they are manufacturing CPU's consisting of multiple processors to keep up with performance demand. However programming for multiple CPU's requires a concurrent programming approach which is quite different to normal sequential programming.

There is a lot of research being done about concurrent programming practices. We relate to two concurrency models which we deem relevant for our needs. The first paradigm is Message Passing[14] which synchronizes by the simple fact that a message can only be received after it has been sent.

The Actor Model[5] is another fundamental concurrency model which treat "Actors" as the universal primitives of digital computation. It utilizes Message Passing as Actors can send and receive messages.

Previous research[15] within our university and together with partners was into distributed computing for creative practices. This research also related to both the Actor Model and Message Passing. From this research a prototype protocol and corresponding orchestrator tool called ZOCP[16] was developed. Our current research is much alike and might ideally merge in a later stadium.

A didactic model

Computational Thinking is considered a 21st century skill[17] and positioned as one of the major skills students need to be taught. One of the best approaches to learning computational thinking is by learning to program[18]. When one wants to operate on technology one eventually runs into programming. However picking up programming is not something one just picks up like that. It is often regarded as very challenging with steep learning curves.

Looking at the process of learning to get acquainted with technologies we have divided four distinct phases a practitioner runs into. *(Note; this is a non academically verified classification based on our experience with dealing with students and technologies in creative practices)*

User	The first phase, in which people use technology. The User only consumes technology and is not able to build significantly new applications with it.
Operator	When one wants to create something with technologies one becomes an Operator. In this level one wants to control and mix technologies and can do so in a limited fashion using pre-built tools and applications.
Scripter	When one is too much limited within the bounds offered by the currently available technology one seeks further control of technologies and can be considered Scripter. In this level one will program and mix technologies.
Developer	When one controls technologies fully one can be considered a Developer. One is only limited by technology itself and their creative mind.

These User Levels support us in creating a didactic model to expose to new users. We want to expose a new user to technologies in a way that supports them in all levels they will reach. We often find that visual programming is very welcoming for new users. We also noticed that these environments are often a dead end when one wants to go further. Either one needs to move to more general programming practices and leave the current practice behind or one ends up building extensions for the tool they already know using its API or SDK.

Ideally we would provide students with a tool that enables them to delve deeper into next user levels without the need to move to other frameworks or tools. A tool like that would consist of visual programming methods as well as classical text based programming. This tool would then also be able to utilize multiple processors which could be accomplished by embracing the Actor Model.

Existing text-based creative coding environments such as Processing and OpenFrameworks use a similar application loop. One implements 'setup' for initialization and 'update' and 'draw' methods for the loop. This is the basis for both frameworks. The Arduino environment does a similar approach with a setup and loop method.

This model is perfectly applicable to our Actor Model as well, with the remark that we only need to add receiving messages. We can mentally consider every actor to be a virtual Arduino which is able send and receive messages.

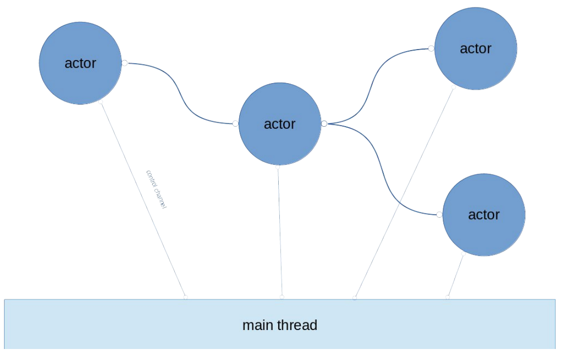
Since the Arduino is very close to plain C development and OpenFrameworks to C++, we can design a library with these approaches that could be deconstructed into libraries used within existing practices.

Implementation

For this initial phase of development we wanted to create a low level software layer which we could build on top of. We wanted to assure the technologies we chose now would be relevant once we improved on the first development phase. For our prototype we chose to develop in C because we wanted to be able to use it in other frameworks and programming languages later on. We find C easiest and most mature to bind to other programming languages.

We have determined the following requirements: We needed concurrency primitives as actors need to able to run on any processor available simultaneously. We chose to use a thread per actor. Actors need to communicate with each other therefore we need thread safe datastructures for communication. We found Zeromq's czmq[19] framework to provide the primitives we needed we needed and more. It also enabled us to build upon its provided actor methods.

Every actor runs a reactor pattern[20] which utilizes a kernel level polling mechanism. This way an actor can block on multiple communication channels. Each actor has a dedicated in memory point to point communication channel for control and uses a publication (pub) socket to publish data to other actors. An actor can subscribe to another actor's "pub" socket through a subscribe (sub) socket. Using this topology actors can communicate with each other and can be managed. The following diagram illustrates the actor setup.



With this setup actors trigger on events on its sockets. We have added a simple timeout mechanism which enable actors to also operate on timed triggers. Through this an actor can for example also use a typical game loop programming model used in OpenFrameworks and Processing. We expect this to be welcomed by users coming from those frameworks.

The GUI application, which we have dubbed *Gazebosc*[21], is a very minimal SDL application using an Immediate Mode UI to represent the graph of actors. It provides control of the actors, its connections and is able to spawn and destroy actors. It can save the graph of actors to file and load them from a file.

We want a visual representation of all actors using typical nodal logic. However since we run completely concurrently it is not safe for actors to draw a graphical representation by themselves. We use the dedicated control communication channel to report its status which we use to draw a GUI. The exact implementation of this Report API is ongoing. For now it reports a string representation of the actor's state.

Because the Report API used for the graphical representation is an asynchronous communication channel it cannot represent the exact current state of the actor. When a report is received, the actor might already be in a next state. For example it can be the case that this specific actor processes much faster than we can represent graphically. This is an inherent feature of concurrent programming and needs to be accounted for by the user.

To enable easy creation of Actors we embedded Python into the prototype. Python has the unfortunate limitation of having a Global Interpreter Lock which limits concurrent computation to sequential limitation. As performance is not our primary focus, this is no problem for our current prototype. We are closely following Python's sub-interpreter[22] support which would overcome these limitations.

The development of the prototype is in its early stages. Many things can be said about about its design. For example one could question whether it is optimal to use a thread per actor and let the Operating System schedule it. Microsoft Windows operates on a different IO model and uses a 'Proactor' pattern therefore we expect performance on Windows to be sub-optimal. However our questions are currently focused on whether we can let novice users operate on this model. We will later focus on optimizing its performance and platform support.

Initial feedback test

We organized a Freaklab[23] in which we have exposed some users to the initial prototype. We were interested in their feedback about the concurrency challenges, whether they could build and run the prototype on their computer and whether they would be able to implement a simple HelloWorld actor or Python actor.

We invited people from our network which we can classify as being 'developers' in our user-level classification. We explicitly needed developer level users because of the early stage of the prototype. We expect challenges in building the prototype and tracing bugs using debuggers on different platforms.

Results

The Freaklab session was held with 8 people attending using OSX and Linux systems. As we expected mostly technical feedback we list some of the issues we encountered

The initial hurdle to overcome were building issues on some versions of OSX. These hurdles are important for us to eradicate as these often are discouraging for new users. Our session was used to find causes of these issues. We also noticed GIL errors when using Python before version 3.7. We anticipate more GIL challenges once we create more actors due to the concurrent nature of the prototype. Linux worked out of the box.

We did not test Windows as there were no people using Windows. We have also not succeeded in building the prototype on Windows due to the different build environments used by dependencies. This needs to be sought out first.

After users were able to run the unit tests and build the GUI application we moved on to building custom actors. The easiest approach to building custom actors is using the embedded Python interpreter. This was picked up by the users and gave no trouble. The hardest part was in returning an OSC message from Python. Currently we provide no methods for constructing or deconstructing OSC messages. This clearly needs addressing to make this more easy.

Another approach to creating custom Actors is by using C++. The GUI app included a GActor class which can be inherited. This took quite some time for users to accomplish and we noticed from this exercise that the C++ API needs to be simplified. Users now need to explicitly use the same name for declaration and instantiating. This make sense from a compiler and runtime perspective but would be easier if this is handled more automatic. In essence the C++ API should work similar to the Python API which just dictates a name for the handler method inside a class.

Another issue we noticed was the in naming of our API. We interchangeably use "node", "actor", "backend" and "fronted" which is confusing. The Sphactor API uses specific methods to control the running thread(actor or node) from the main thread. This is very important to do right, for example C allows us to run these methods from inside the running threads as well which could cause errors as this is not thread safe. The API should support the user not to do so. It would be helpful if the naming of the API methods help the user to know from what context it is calling a method. One suggestion was to use a mental model of "asking" an actor. This would be helpful in understanding that we are asking from a second person perspective an actor to do something.

In a later attempt we have tested a scenario using a Python Actor to play audio trigger by a "Manual Pulse" actor, so that a manual trigger on different buttons would trigger audible sounds. We refer to setups like this as a "stage". The stage can be saved to a file. We loaded this saved stage file onto an other computer running a different platform. This worked flawlessly and illustrated how we want to enable cross-platform support. In this scenario we benefit from Python's cross platform support. This scenario also demonstrated the latency of actor communication as no audible delay was noticeable. We still need to acquire real performance metrics of the actor's communication. This will be done in a later iteration of development when we focus more on the performance of our framework.

Conclusion

The design of our prototype has shown some of its potential during the tests we performed. We have succeeded in getting the prototype to run on other user's machines and in getting users to extend the possibilities of the prototype by creating custom actors. We acquired feedback from users during our Freaklab session which we use in our next development iteration. Although we do not have any performance metrics yet we are confident to continue development in the direction we're heading. As our current focus is not on performance but rather usability our next iteration will be in simplified API's, custom Actor creation and more general usage example Actors.

If in a later stadium we need to enhance performance we have a very good option to minimize file descriptor and thread usage by merging actors to utilize a thread pool. The control communication channel will then be reduced to one channel per CPU core. This can reduce context switching overhead and file descriptors usage. Further optimization can then also be accomplished by enabling actors which operate on the same thread to just call each other's method instead of through their pub-sub sockets. From a mental model this should make no difference as actors still send and receive messages. How it's handled in the backend should be of no concern to the user. For now we leave this open as an exercise for a later moment.

References

1. Wright, Matthew. *Open Sound Control 1.0 Specification*. 2002, http://opensoundcontrol.org/spec-1_0.
2. Maeda, John. *Creative Code*. Thames & Hudson, 2004.
3. *What Is Max? | Cycling '74*. https://cycling74.com/products/max/. Accessed 29 Jan. 2020.
4. *Pure Data — Pd Community Site*. https://puredata.info/. Accessed 29 Jan. 2020.
5. Agha, Gul A. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
6. *TROIKATRONIX : ISADORA*. https://troikatronix.com/. Accessed 29 Jan. 2020.
7. *Touchdesigner*. Derivative, https://derivative.ca/. Accessed 29 Jan. 2020.
8. *Processing*. https://processing.org/. Accessed 29 Jan. 2020.
9. *OpenFrameworks*. https://openframeworks.cc/. Accessed 29 Jan. 2020.
10. Witters, Koen. *DeWiTTERS Game Loop – DeWiTTERS*. https://dewitters.com/dewitters-gameloop/. Accessed 29 Jan. 2020.
11. Mazza, Emanuele. *Mosaic*. 2018. 2020. *GitHub*, https://github.com/d3cod3/Mosaic.
12. *Ossia – Open Software System for Interactive Applications*. https://ossia.io/. Accessed 29 Jan. 2020.
13. Sutter, Herb. "A Fundamental Turn Toward Concurrency in Software."*Dr. Dobb's*, http://www.drdobbs.com/web-development/a-fundamental-turn-toward-concurrency-in/184405990. Accessed 29 Jan. 2020.
14. Andrews, Gregory R., and Fred B. Schneider. "Concepts and Notations for Concurrent Programming." *ACM Computing Surveys*, vol. 15, no. 1, Jan. 1983, pp. 3–43. *DOI.org (Crossref)*, doi:10.1145/356901.356903.
15. Loonstra, Arnaud. *Concurrency for Creative Coding*. Media Technology MSc Programme - Leiden University, 8 Oct.

2015, https://mediatechnology.leiden.edu/research/theses/concurrency-for-creative-coding.

16. Loonstra, Arnaud. *Orchestrating Computer Systems, a Research into a New Protocol*. http://z25.org/static/_rd_/zocp_init_plab/index.html. Accessed 29 Jan. 2020.
17. Jenkins, Henry. *Confronting the Challenges of Participatory Culture: Media Education for the 21 St Century*. 2006.
18. van der Maas, Han. *Computational Thinking – 'Op Een Creatieve Manier Problemen Oplossen*. 30 Nov. 2016, https://www.kennisnet.nl/artikel/computational-thinking-op-een-creatieve-manier-problemen-oplossen/.
19. *CZMQ High-Level C Binding for ØMQ* . 2011. The ZeroMQ project, 2020. *GitHub*, https://github.com/zeromq/czmq.
20. Schmidt, Douglas C. *Patterns for Concurrent and Networked Objects*. Wiley, 2000. *Open WorldCat*, http://www.vlebooks.com/vleweb/product/openreader?id=none&isbn=9781118725177.
21. *Gazebosc*. 2019. HKU Expertise Centre Creative Technology, 2020. *GitHub*, https://github.com/hku-ect/gazebosc.
22. "PEP 554 -- Multiple Interpreters in the Stdlib." *Python.Org*, https://www.python.org/dev/peps/pep-0554/. Accessed 29 Jan. 2020.
23. Loonstra, Arnaud. *Freaklabs: Joint Artists and Developers Technology Design and Evaluation*. http://z25.org/static/_rd_/freaklab_plab/index.html. Accessed 29 Jan. 2020.