Further exploration with ACTORKIT: an Actor Model of Concurrency

Kwabena Aning
Department of Computer Science and Information Systems
Birkbeck, University of London
London, WC1E 7HX, UK
Email: k.aning@dcs.bbk.ac.uk

Keith Leonard Mannock
Department of Computer Science and Information Systems
Birkbeck, University of London
London, WC1E 7HX, UK
Email: keith@dcs.bbk.ac.uk (Contact Author)

***Keywords*—Distributed computing, Design and implementation, Software Architectures, Parallel and Concurrent Computing, Agent Architecture, Programming languages.**

***Abstract*—In this paper, we describe further explorations with our prototype architecture and implementation of the *Actor* model[1] of concurrent computation, ACTORKIT[2][3]. Actors provide a mechanism to exploit the multi-core processors of modern day computer architectures, in an intuitive and natural fashion. ACTORKIT is hosted in an existing programming language as native constructs (Swift). We describe some of our extensions to the language and our experiences with the development, and evolution, of our prototype implementation.**

I. INTRODUCTION

With the inherent limitations of current processor technology there has been a growing movement towards the use of processors with multiple cores and the use of multiple processors. The aim of these processor architectures is to improve the throughput, efficiency, and processing power of the computer but these benefits do not come without their own problems and challenges. There are many challenges when building software that can leverage these capabilities, while still enabling a tractable programming model. One consequence is that this can lead to quite low-level constructs which utilise shared resources, to enable the promised processing power with multi-core computing.

In the Section II, we will consider the main differences between concurrency and parallelism as there is often confusion over these terms. In Section III we outline the rationale for our usage of the Actor model of computation. We also consider the problems we wish to address that occur with concurrent programs. Section IV describes the Actor model and its key components. The following section, describes the architecture of our prototype system and one of its implementations.

In Section VI we assess our current prototype in terms of the functionality, reliability, ease-of-use, and performance, when considering solutions to a set of computer science problems. Section VII provides a brief review of the main work that is related to this topic. We conclude the paper with a discussion of future work.

II. CONCURRENCY AND PARALLELISM

To provide a context to the rest of this paper we need to draw a distinction between concurrency and parallelism. Concurrent programs are best described as an interleaving of sequential programmes[4]. A concurrent program has multiple logical threads of control. These threads may or may not run in parallel [5]. This distinction is depicted graphically in Figure 1.

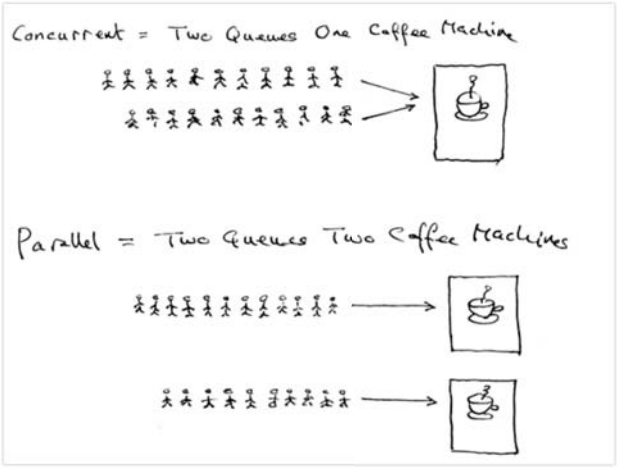


Figure 1: Concurrency vs Parallelism (© Joe Armstrong)

It is also important to stress that there is an *order* aspect to the definition of concurrency. Tasks can be performed in any order, and this allows for parallelism as the tasks can then be shared between several processes if the order that they are performed is irrelevant.

Parallelism may be seen as an inherent benefit of concurrently written programs. A parallel program is one whose tasks can be distributed across more that one process. This does not imply that the program is working on different tasks at once. It simply states that the program is written in such a way so that different parts of it, or its computations, can be run, or can be performed, on different processors simultaneously. For example, writing the execution logs of a program to the file could be executed on one processor, while reading the file could be done on another. This is possible because the two processes can run independently of each other.

A small selection of the typical models of concurrency and parallelism include: a) Actors, b) Shared memory (mutex, Software Transactional Memory)[6], c) Communicating Se-

quence Processes (CSP)[7], d) Futures/Promises, e) Coroutines and Continuations [8], f) Event based IO and multiplexing [9]. An in depth comparison of these different methods will be presented in a future paper but [10] provides a brief comparison.

We have focussed on the Actor model utilising *coroutines* for one of our implementations.

III. "Why Actors?"

When more than one process requires access to a shared resource one has to make certain decisions about the granularity of access. The coarser the access the less concurrency that is possible (in general). The standard approach to such a problem is to either incorporate features into the programming language, e.g., threads in Java, or leverage a *toolkit* or library. These low level constructs can lead to extremely complex interactions between the various processes leading too contention, with the possibility of deadlock, race-conditions, etc. [11] We can adopt *higher-level* features, built upon these lower level constructs, for example the `java.util.concurrent` library, but these still have the same underlying issues and a *learning curve* associated with them.

Some programming languages have focussed on these problems, e.g., Erlang, but, by and large, they have not transitioned to the mainstream (having said that popular application WhatsApp, uses Erlang) [12]. Other programming languages have also considered these issues, e.g., *Pony* [13], *Go* [14], *Scala* [15] , and *Clojure* [16], amongst many, with varying levels of success.

As there are many different approaches to dealing with concurrency in modern programming languages, what makes the actor model an appealing one? First let us consider a number of potential *pain points* in achieving concurrency.

a) Mutable state is hard: One of the main problems with concurrency is shared mutable data. If two different threads have access to the same piece of data then they could try to update it at the same time. This can lead to inconsistent data, either different versions of the data, or the data being corrupted.

The standard method to avoid these problems is to use locks to prevent data updates from occurring simultaneously. This can cause performance problems and, most of all, is very difficult to write, which can lead to "buggy" software.

b) Immutable data can be safely shared: Any data that is immutable (i.e., it cannot be changed) is safe to use concurrently, as it can never be updated; is is updates of shared state that cause most problems in concurrent software.

c) Isolated data is safe: If a block of data has only one reference then it is known as *isolated*. Isolated data cannot be shared by multiple threads, and therefore there are no potential concurrency problems. Isolated data can be passed between multiple threads as long as only one of the threads has a reference to it at any one time, then the data is still safe from concurrency problems. By sharing only immutable data, and exchanging only isolated data, we can have safe concurrent programs without resorting to locks.

d) Every actor is single threaded: The code within a single actor is never run concurrently. This means that, within a single actor, data updates cannot cause problems.

The problem is that it is very difficult to do all of these correctly, especially when considered isolated data. If you retain a reference to some isolated data you have passed to another actor/thread, or change something youve shared as immutable, then the system will become unstable, and inherently unsafe. What is required are language level constructs that enforce these constraints for the programmer.

Several languages address some of these points, but our approach has a number of advantages:

- leveraging an existing, *main stream* programming language,
- established tooling support (thought Xcode[17] or AppCode),
- utilisation of existing, high quality, third party libraries and frameworks, and
- a well established compiler and runtime chain (through the LLVM compiler infrastructure[18]).

Our aim has been to provide an elegant solution to concurrent programming problems, specifically, we are guided by the principles of simplicity, consistency, performance, and completeness.

- Simplicity — It is important for the interface to be simple and clear to use. The underlying system implementation should hide the awkward, or troublesome, code. The faster the programmer can get to work, the more productive they will be.
- Consistency —- The formalism that is used should match the approach taken in the host programming language; it should be a *first class citizen* within the language rather than an obvious add-on.
- Performance — Runtime speed and efficiency are important but *correctness* is the priority; if the program produces the wrong result, or hangs, then all the optimisation is wasted.
- Completeness — The formalism should provide solutions for the majority of concurrent problems that can be addressed. This should include functionality that achieves the same as lower-level, more problematic, solutions.

The actor model is a conceptual model to deal with concurrent computation. The model defines some general rules for how the systems components should behave and interact with each other. The most well known current day implementations of this model are the Erlang [19] and Elixir [20] programming languages, or the Akka framework for Scala and Java on the JVM [21]. We believe that the Actor model is the best approach to addressing the requirements noted above.

IV. The Actor Model

Actor models have the following key characteristics: Actors, fault tolerance, and distribution. The key aspect of the model is that Actors communicate with each other by sending asynchronous messages, and those messages are stored in other actors' mailboxes until they are processed.

A. Actors

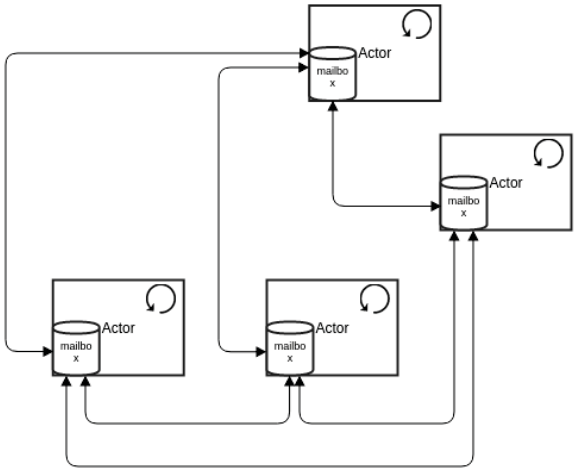
An actor is the primitive unit of computation in this model. It receives a message and then, based upon the type of message, performs some computation.

The idea is very similar to what we have in object-oriented languages where an object receives a message (a method call) and does something depending on which message it receives (which method we are calling). The main difference is that messages in the Actor model are asynchronous and that actors are completely isolated from each other. Actors can have private state but they never share memory and cannot be modified directly by another actor.

Actors do not operate in isolation; they come in systems. In the actor model everything is an actor and they have addresses so one actor can send a message to another, and a control system to manage these interactions. This enables multiple actors to run at the same time, but each actor will, in, and of itself, process a given message sequentially. If one sends three messages to the same actor it will process them one at a time, one after the other; there is no concurrency within the actor. To have these three messages executed concurrently one needs to create three separate actors, and send one message to each of them. Of course, this may result in different behaviour as each actor will now have its own, private, internal state.

Messages are sent asynchronously to an actor and therefore each actor needs to store those messages while processing another message. The *mailbox* is the place where these messages are stored.

The following figure illustrates actors interacting with each other, each of which autonomously runs in its own process signified by the circular arrow. Each actor has their own attached mailbox, to which each of the actors can send messages.



Basic actors

Actors are restricted to the following actions upon receiving a message:

- Create more actors
- Send messages to other actors
- Change its state (designation)

The final point indicates that while the system consists of passing around immutable messages, the internal state of an actor *is* mutable. For example, if we consider an actor emulating a simple calculator. Its initial state is the number 0. Upon receiving the message `plus(1)`, instead of simply mutating its state it indicates, designates, that for the next message it receives, the state will be 1.

Therefore, each individual actor has:

- An internal state, which is only mutable by itself.
- A mailbox into which it receives messages.
- An internally accessible method for interacting with those messages.
- An implementation of a protocol that allows it to communicate with other actors.

B. Fault tolerance

One of the key aspects of a concurrent system is fault tolerance; a established, and accepted means to deal with situations when things go *wrong*. Erlang introduced the "let it crash" philosophy[19]. The idea is that you should not need to program defensively, trying to anticipate all the possible problems that could happen and find a way to handle them. What Erlang does is let the process fail (crash), but make this critical code be supervised by another process whose only responsibility is to know what to do when this failure occurs. For example, this could mean resetting this part of the code to a stable state, and is fundamental to the operation of the actor model.

This architectural model leads to, so called, *self healing* systems, where if an actor receives an exceptional state and crashes, for whatever reason, a supervisor can address the situation, returning the system to a consistent state.

C. Distribution

Another aspect of the actor model is that it should not matter if the actor that Im sending a message to is running locally or in another node. An actor is just a unit of code with a mailbox, internal state, and a supervisor, so the actual location of the actor is irrelevant (apart from performance considerations). This also enables *hand off*, where an actor can be migrated from one processing location to another; in the simplest form, all that is required, is to modify the "address" of the actor.

Overall, the advantages of the Actor model are:

- Utilises message passing and channels,
- No shared state (avoids locks, easier to scale),
- Easier to reason about the code (maintenance).

Some perceived disadvantages might include:

- When shared state is required,
- Implementing shared data can be simpler to implement for straightforward use cases,
- Handling large messages, or a lot of messages,
- Messaging is essentially a copy of shared data.

V. The Architecture of ActorKit

Our architecture is based upon an enhancement to the Swift programming language as: a) it is open-source, b) it is efficient, c) it isn't *yet another programming language* which hardly anyone will use, d) it is compiled, e) it is available on several platforms (outside of the Apple ecosystem), f) it is an object-oriented language with functional programming paradigm features, and g) it is growing rapidly in popularity. By adopting a mainstream language with less *baggage* than existing languages we get the best of multiple worlds; a potentially large user community, and a reliable and efficient programming language. The success of Actor frameworks such as Akka for Scala and Java suggested a way forward.

The basic architecture and implementation of our system have been presented in [2] and [3]. The overall architecture is shown in Figure 2.

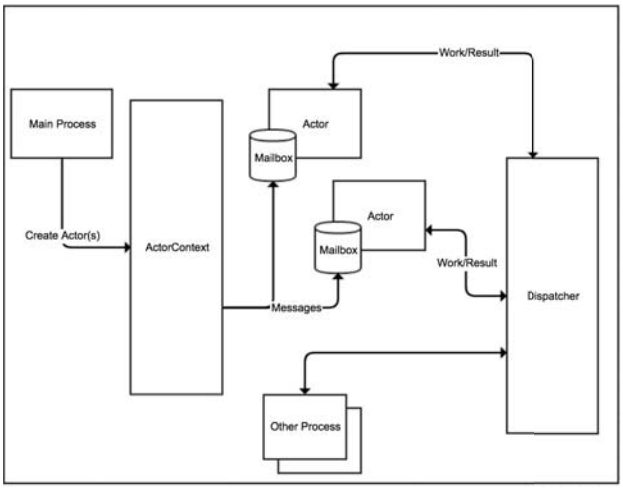


Figure 2: The architecture of ActorKit

One restriction of the Actor model is that an actor can only be created within a *context*; this is to ensure that it has all the necessary properties to communicate with other actors (within the current context).

As this work is an extension of the Swift programming language, the *Grand Central Dispatch* library [22] is utilised for the MacOS implementation. We have refined this implementation since our original prototype [2], and our alternative implementation of the language, based on the open source version of Swift, can run on other platforms. The key to this implementation is to leverage the equivalent of *coroutines* in Kotlin.

We will now briefly describe the main components of our model, and then discuss the alternative implementation, informally known as the *coroutine implementation*.

The three key elements of our model are: the *Actor Context*, *the Actor*, and the *Mailbox*.

A. The Actor Context

This is the logical domain for the creating and running of actors. The context defines a *namespace* for actors, allowing actors within that same context to communicate with each other. The context is also responsible for creating the actors as they cannot, and should not, be instantiated in isolation. This aspect is implemented using the `protocol` feature of the Swift, enforcing a *contract* on any sub-types of `ActorContextProtocol`. The context is responsible for keeping track of all actors created within it, and it also keeps track of the mailbox assigned to each actor. Once actors have been removed from a context the orphaned mailboxes can be reassigned to actors that are created to replace them, thus providing a *pool* of contexts.

B. The Actor

This is the main unit of computation. The actor defines the means to process messages it receives and any other business logic associated with the work that it does. The actor is attached to, but does not own, a mailbox from which it receives its messages (that is the responsibility of the context). Our actor implementation within Swift is carried out in a *type safe* manner so that message types are clearly defined. This has the added advantage of predictable message handling on the side of the Actor.

The actor has methods for "telling", or sending messages to other actors. The `processor` method takes a message and returns nothing and is the component of the actor that does the work parcel allocated to the actor. The processor is invoked on a separate thread using the *Dispatch Framework* [22], for the OSX version, and via a *manager* process for the coroutine implementation. The actor notionally *polls* the mailbox attached to it to ascertain whether it has messages or not; in reality, this is implemented as an event feedback loop, in a manner similar to coroutine dispatch in Kotlin and the dispatch loop of `Node.js`. When the actor receives a message it invokes the defined processor on a separate thread with the message it has just received from its mailbox.

C. Mailbox

The Mailbox could also be referred to as a *message queue*, which is attached an actor. Actors never directly receive messages as the *Actor Context* is responsible for routing messages to the given actors mailbox, and then the actor retrieves the message from its attached mailbox. This ensures that, should an actor stop working for any reason such as entering into an exceptional state, or receiving a termination message from its supervisor, messages that are sent to that actor while it is shutting down are not lost. These *orphaned* messages are effectively *buffered* in the mailbox, waiting for the next actor to take control.

This component is implemented as a simple typed queue. A collection that accepts and provides an API for storing and retrieving homogeneous messages.

D. Coroutines

As mentioned earlier, one of the more interesting developments in this work is the port to a non-OSX dependent version of ActorKit. To achieve this we have tried multiple different

approaches but the one we wish to discuss here is a coroutine one. A coroutine can be considered to be a function that has the ability to pause its execution, pass control to another coroutine, and then to continue where it left off when control returns to it. It can be seen in many modern programming environments but one *use case* is that of computer gaming, and coroutines are part of the Unity library for 2D and 3D games[23].

For our problem, which can be viewed as a state-based computation, coroutines are naturally appealing as they make it much easier to express, and easier to understand, the underlying problem. To this end we have leveraged the coroutine facilities of the Kotlin programming language, developed by JetBrains, a main language for the Android platform, but that also can target native environments. By utilising their approach we have can provide operations that can be *suspended* and *resumed* at a later time, potentially using a different thread of execution. This has led us to develop a library which we can use with the LLVM, our base implementation toolkit, to provide the functionality for our actor system.

VI. Evaluation

To assess the functionality, reliability, ease-of-use, and performance characteristics of our latest version of ActorKit we have considered a number of different computer science problems, known for their suitability for concurrent computation[24]. We have taken some of those problems and then implemented them using ActorKit, Akka, Go, Kotlin, and Erlang. Full details of these comparisons will be detailed in a future paper but some overall conclusions of this evaluation can be presented here. (In the following discussion, when we refer to *language* please read that as *programming language* or *framework*.)

Each of the problems were coded in each language and then compared under the four characteristics:

- functionality — does the solution meet the functional specification.
- reliability — does the language enable the corner cases of the solution to be handled in an elegant and natural manner; does the program run consistently and with understandable characteristics.
- ease-of-use — how well do the primitives of the language match the problem and how easy is it to development an appropriate solution.
- Performance — how performant is the program in terms of memory usage, scalability, cpu utilisation, etc.

A selection of the problems we addressed are: a) producer-consumer, b) generation of π, c) common factors, d) prime number generation, and e) fibonacci numbers. In general, the family of *divide-and-conquer* algorithms are are best suited to our formalism.

These problems were given to a group of programmers (20), who were (reasonably) fluent in all of the various languages in the study. They were then asked to code appropriate solutions to the problems. No time limit was applied to these exercises, other than the cutoff of the study period (four weeks). Aside from the empirical results, we were more interested in their feedback on the functionality, and ease-of-use, of each of the languages. Summarising our findings, they, were general impressed with the ease-of-use of ActorKit, and the functionality that it affords. Erlang and Go came out as leaders in this study, with Akka, due to the complexity of setup, trailing behind the others. From a reliability viewpoint, ActorKit was seen to be dependable but its performance was slowest of the various alternatives. Its memory usage and cpu utilisation showed up that we need to do additional work on the optimisation of the code produced by our coroutine module.

VII. Related Work

As stated previously, our work builds upon the Actor model but we also owe some concepts and motivation to a number of other technologies and programming languages. In [2] and [3] we have provided a review of these but the remainder of this section briefly describes the salient points from those essential languages and frameworks.

A. Erlang

The Erlang Virtual Machine provides concurrency for the language, in a portable manner and as such it does not rely to any extent on threading provided by the operating system nor any external libraries. The self contained nature of the virtual machine ensures that any concurrent programmes written in Erlang run consistently across all operating systems and environments.

The simplest unit in the language is a lightweight virtual machine called a *process* [25]. Processes communicate with each other through *message passing*. A simple process written to communicate between processes could be:

```
start() -> spawn(module_name, [Parameters]).
loop() ->
 receive
   pattern -> expression;
   pattern -> expression;
   pattern...n -> expression;
 end
loop().
```

`start()` spawns the process for the current module with any parameters that are required. A loop is then defined which contains directives to execute when it receives messages of the enumerated patterns that follow — `loop()` is then called so that the process can, once again, wait to receive another message for processing.

The above will code fragment will exhibit the behaviour pattern below:

$$S * E \rightarrow A, S\prime \tag{1}$$

Therefore, given a *state* `S` with an occurrence of an *event* `E`, some *action(s)* `A` should be performed that transitions our process to a new *state* `S`$\prime$. In this case *expression* is the representation of the transition of the program from one state to the other.

Erlang in itself provides several constructs for writing concurrent programmes in ways that allows for runtime optimisation and fault tolerance. Capabilities such as *hot code swapping, links, monitors, supervisors, timeouts* and so on are all built in capabilities available to an Erlang programmer. These have to be coordinated manually to achieve the expected result. The OTP framework provides a library for grouping these error-prone manual processes into well tested and coordinated best practises and standards. OTP does most of the heavy lifting for the developer interested in concurrency and also providing minimalistic boilerplate code for required behaviours of generic applications. The OTP library is distributed with all modern versions of the language environment and as such can be considered as part of the Erlang standard distribution. Behaviours, or *interfaces*, are defined, but it is up to the developer to provide the business logic within the applications that use any generic behaviours. The OTP library also provides best practices for structuring Erlang code.

B. Pony

Pony is an object-oriented, actor-model, capabilities-secure programming language [13]. In object oriented fashion, an actor designated with the keyword *actor* is similar to a class except that it has what it defines as *behaviours*. Behaviours are defined as asynchronous methods defined in a class. Using the *be* keyword, a behaviour is defined to be executed at an indeterminate time in the future.

```
actor AnActor
    be(x: U64) =>
        x * x
```

Pony runs its own scheduler using all the cores present on the host computer for threads, and several behaviours can be executed at the same time on any of the threads/cores at any given time, giving it concurrent capabilities. It can also be viewed within a sequential context also as the actors themselves are sequential. Each actor executes one behaviour at a given time.

C. The Akka Library

Earlier versions of Scala had natively implemented actors as part of the Scala library and could be defined without any additional libraries. Newer versions (2.9 and above) have removed the built in *Actor* and now utilise the Akka Library.

Akka is developed and maintained by LIGHTBEND [26] and when included in an application, dependable concurrency can be achieved. Actors are defined as classes that include or *extend* the *Actor* trait. This feature enforces the definition of at least a `receive` function, which is defined as a partial function, taking another function and returning a `Unit` (void value).

The function `receive` takes as a parameter is the behaviour that the developer needs to program into the actor. This is a essentially defined as a pattern matching sequence of actions to be taken when a message is received that matches a given pattern.

```
import akka.actor.Actor
class MyActor extends Actor {
    def receive = {
            case Message1 =>
                    //some action
            case Message2(x:Int) =>
                    // another action use
                    // x as in int
            ...
            case MessageN =>
                    //Other actions
    }
}
```

At the heart of the Akka Actor implementation is the Java concurrency library `java.util.concurrent` [27]. This library provides the *(multi)threading* that Akka Actors use for concurrency. Users of the library do not need to worry about scheduling, forking and/or joining. This is dealt with by the library's interaction with the executor service and context.

The Akka Library offers options to select which *executor service* to use. It currently defaults to the *ForkJoinPool*, which is usually sufficient for most tasks. This is referred to as the *Dispatcher* [26][28] and is equipped with the functionality to determine the execution strategy for a given program, such as which thread to use, how many to make available in a pool for actors to run on, etc.

All these options are exposed in a malleable *configuration factory*. Developers are able to fine tune the actor system's behaviour which relative ease.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we have briefly described our further explorations with our prototype architecture and implementation of the *Actor* model of concurrent computation. ACTORKIT is hosted in the Swift programming language as native constructs. We have described some of our extensions to the language and our experiences with the development, and evolution, of our prototype implementation. The generalisation of out implementation, to a cross-platform solution, has involved the development of various approaches to the thread management problem, the coroutines approach being both novel, and, with appropriate refinement, performant.

Further experimentation is required with the coroutine implementation, and it is far from performant in its current version, but it is a fruitful area of research. We will be developing a more rigorous set of experiments with which to evaluate our model and implementations, and, once further optimisation has taken place, we will be able to make a more analytic comparison of the various options. We have added an *event bus* which can add further scheduling facilities, logging mechanisms, and further monitoring capabilities. This has enabled us to monitor the various actor systems and their interactions, although this aspect of the research needs further refinement and development.

We have already extended the model to operate in a distributed environment via the use of physical processors, cloud

solutions, and virtual containers (Docker images). While this work is still in its infancy the initial results are most promising.

References

[1] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence", in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[2] K. Aning, Kwabena; Mannock, "An Actor Model of Concurrency for the Swift Programming Language", in *International Conference on Software Engineering Research and Practice (SERP'17)*, 2017, pp. 178–183.

[3] ——, "An architecture and implementation of the actor model of concurrency", in *Information, Intelligence, Systems & Applications (IISA)*, 2017, pp. 1–6.

[4] J. Reppy, C. V. Russo, and Y. Xiao, "Parallel concurrent ML", *ACM SIGPLAN Notices*, vol. 44, no. 9, p. 257, 2009, ISSN: 03621340. DOI: 10.1145/1631687.1596588.

[5] P. Butcher, *Seven Concurrency Models in Seven Weeks: When Threads Unravel*, 1st. Pragmatic Bookshelf, 2014, ISBN: 9781937785659.

[6] N. Shavit and D. Touitou, "Software transactional memory", *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[7] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, International Series in Computer Science, vol. 21, no. 8, R. M. McKeag and A. M. Macnaghten, Eds., pp. 666–677, 1978, ISSN: 00010782. DOI: 10.1016/0167-6423(87)90028-1.

[8] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines", *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 2, pp. 1–31, 2009, ISSN: 01640925. DOI: 10.1145/1462166.1462167.

[9] P. Haller and M. Odersky, "Event-based programming without inversion of control", *Modular Programming Languages*, vol. 4228, no. 1, pp. 4–22, 2006, ISSN: 10990798. DOI: 10.1016/j.semradonc.2011.02.001.

[10] B. Cantrill and J. Bonwick, "Real-world Concurrency", *Queue*, vol. 6, no. 5, p. 16, 2008, ISSN: 15427730. DOI: 10.1145/1454456.1454462.

[11] B. Shao, N. Vasudevan, and S. A. Edwards, "Compositional deadlock detection for rendezvous communication", *Proceedings of the seventh ACM international conference on Embedded software - EMSOFT '09*, p. 59, 2009. DOI: 10.1145/1629335.1629344.

[12] WhatsApp, *WhatsApp :: Home*, 2015.

[13] S. Clebsch. (2015). The Pony Programming Language, [Online]. Available: http://www.ponylang.org/.

[14] R. Pike, "Go at Google", in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12, New York, NY, USA: ACM, 2012, pp. 5–6, ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384720.

[15] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 2nd, 2-3. Artima Inc, 2011, vol. 410, ch. 2, pp. 202–220, ISBN: 9780981531618. DOI: 10.1016/j.tcs.2008.09.019.

[16] R. Hickey, "The Clojure Programming Language", in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS '08, New York, NY, USA: ACM, 2008, 1:1–1:1, ISBN: 978-1-60558-270-2. DOI: 10.1145/1408681.1408682.

[17] Apple Inc, *Xcode IDE*, 2017.

[18] C. Lattner, "Introduction to the LLVM Compiler Infrastructure", *Computer-Aided Civil and Infrastructure Engineering*, vol. 21, no. 5, pp. 319–320, 2006, ISSN: 1093-9687. DOI: 10.1111/j.1467-8667.2006.00438.x.

[19] J. Armstrong, "Erlang", *Communications of the ACM*, vol. 53, no. 9, p. 68, 2010, ISSN: 00010782. DOI: 10.1145/1810891.1810910.

[20] G. Fedrecheski, L. C. Costa, and M. K. Zuffo, "Elixir programming language evaluation for IoT", in *Proceedings of the International Symposium on Consumer Electronics, ISCE*, 2016, pp. 105–106, ISBN: 9781509015498. DOI: 10.1109/ISCE.2016.7797392.

[21] R. Roestenburg, R. Bakker, and R. Williams, *Akka in Action*. 2015, p. 443, ISBN: 9781617291012.

[22] A. Inc. (2016). Dispatch - API Reference, [Online]. Available: https : / / developer . apple . com / reference / dispatch.

[23] Unity, *Unity - Game Engine*, 2017.

[24] M. Raynal, *Concurrent programming: Algorithms, principles, and foundations*. 2013, vol. 9783642320, pp. 1–515, ISBN: 9783642320279. DOI: 10.1007/978-3-642-32027-9.

[25] J. Armstrong, *Programming Erlang*. Pragmatic Bookshelf, 2007.

[26] Typesafe. (Feb. 2014). Build powerful concurrent & distributed applications more easily, [Online]. Available: http://www.akka.io/.

[27] D. Wyatt, *Akka concurrency*. Artima Incorporation, 2013.

[28] T. Lockney and R. Tay, "Developing an Akka Edge", 2014.