View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Birkbeck Institutional Research Online

An Actor Model of Concurrency for the Swift Programming Language

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

***Keywords*—Software Architectures, Distributed and Parallel Systems, Agent Architectures, Programming languages, Concurrent computing.**

***Abstract*—The Swift programming language is rapidly rising in popularity but it lacks the facilities for true concurrent programming. In this paper we describe an extension to the language which enables access to said concurrent capabilities and provides an api for supporting such interactions. We adopt the ACTOR model of concurrent computation and show how it can be successfully incorporated into the language. We discuss early findings on our prototype implementation and show its usage via an appropriate example. This work also suggests a general design pattern for the implementation of the ACTOR model in the Swift programming language.**

I. INTRODUCTION

Modern day computing has transitioned to multicore processing as the standard. Most devices have processors with several cores, or multiple processors for that matter and to this end, there are more options for increased efficiency in computational processes. This increased advantage however comes with a challenge - building software that can harness this added computational power for better efficiency without further exacerbating the existing challenges with multi-core computing is difficult.

The shared resource problem is ubiquitous in environments where concurrent processing is required. As more than one process requires access to a singular resource, decisions have to be made on how that resource is made available to said processes. Programming languages have taken on the challenge either natively, or through toolkits or libraries, by providing mechanisms for developers to create concurrently running software. *Newer* programming like *Pony* [1], *Go* [2], *Scala* [3] ,and *Clojure* [4] languages focus on this challenge from the outset.

We decided to adopt the ACTOR model of concurrency [5] to address these problems and to enhance the Swift programming language with an appropriate API and framework. Why Swift? Swift is growing in popularity and since being open-sourced, it can now be run as a general programming language on a variety of platforms. The success of ACTOR frameworks such as Akka for Scala and Java suggests a way forward for Swift that would enhance the functionality of the language while addressing concurrent processing issues.

In the Section II, we will make a distinction between concurrency and parallelism. In Section III of this paper we describe our architecture that addresses the problems with concurrency, problems arising from non-determinism, deadlocking, and divergence, which leads to a more general problem of shared data in a concurrent environment. In Section IV we outline our implementation, and our message passing strategy as a possible solution to the shared data problem. In Section V we describe a brief example to show the usage of our system API. In Section VI we briefly discuss related concurrency models pertinent to the development of our work. We conclude the paper with a discussion of future work.

II. CONCURRENCY AND PARALLELISM

To provide a context to the sections that follow, a distinction needs to be made between concurrency and parallelism. Concurrent programs are best described as an interleaving of sequential programmes[6]. A concurrent program has multiple logical threads of control. These threads may or may not run in parallel [7]. Tony Hoare expresses this concept of concurrency as follows:

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q \tag{1}$$

Assuming that *P* is a process that involves writing logs to a given file from a given input. *Q* is a process that also involves reading that given file and displaying its contents to a given output. α(P $\parallel$ Q) is therefore all the behaviours that are involved with writing logs to a file, reading logs from a file, and displaying those logs to some output. If our given environment allows for it, any these behaviours can occur at any time, in any sequence. The above equation describes a coming together of these processes where any of these behaviours can be called upon with no contingency.

”A process is defined by describing the whole range of its potential behaviour.“ [8]. In other words, to define a process completely one has to enumerate all the potential behaviours or properties of that process. These properties or behaviours can be referred to in CSP as *alphabets*. Alphabets usually denoted

in the calculus of Communicating Sequential Program (CSP) by α can be described as all the set of names, behaviours, and/or actions that are considered relevant for a particular description of an object or in this case a process.

Given the events in the alphabets of processes αP and αQ respectively, which requires simultaneous execution, P can participate in any or all of its alphabets without affecting or concerning Q and vice-versa, and as such all events are logically possible for this system - a union of all alphabets[8].

It is also important to stress that there is an *order* element to the definition of concurrency. In that 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 does not matter.

Parallelism may be seen as an latent benefit of concurrently written programs. A parallel program is one whose tasks can be distributed across more that one processor. This does not imply that the program is working on different tasks at once. It simply implies 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. Using the illustration above, writing the logs to the file could be executed on one processor, while reading the file could be done on another. In all cases, this is possible because process P and process Q can run independently of each other.

As a more tangible example we consider the Gregory-Leibniz series [9]. The infinite series for calculating π. This series of arithmetic computations can be distributed across different processes and then brought together when they are complete.

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} - \dots \quad (2)$$

the fraction being subtracted can be computed separately and because of the associative nature of addition the results can then be summed in any order as the computations complete. We will elaborate on this further when we discuss an example of the usage of our enhancements to Swift.

III. Architecture

The architecture for our implementation follows a basic architectural pattern that directly corresponds to the *axioms* of the actor model.

An actor based model should exhibit the following properties:

Encapsulation This can be described in terms of modularity. Where the actor encompasses all the aspects of the work it needs to do without the dependency on any other actors or processes.

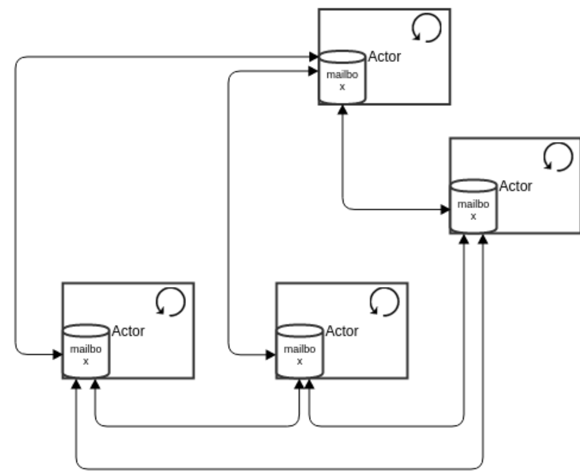
Internal state Only the actor has access to its internal state. It alone can mutate its state. Although it may do so as a result of receiving a message from an external entity, the decision is taken unilaterally to alter variables within itself.

Messaging Actors communicate with each other using messages across a predetermined protocol, and depending on the protocol these messages can sometimes be serialised. Actors can also only send messages to other actors that they know of. The asynchronous nature of the messages sent means that actors need to be designed to continue working even when they have not received responses to messages sent to other actors.

Indeterminacy and quasi-commutativity An order to messages received is not guaranteed. Messages may arrive in any order, and it is the responsibility of the actor to process those messages in a consistent manner.

Mobility and location transparency An obvious benefit of a modular system. A node should be capable of running in a different computational environment (given that the variables for its successfully running are present). An actor should be able to communicate with other actors across different geographical spaces.

The following figure illustrates actors interacting with each other — each of which autonomously runs in it's 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

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.

There are several components to our architectural model which we will now (briefly) describe.

A. The Actor System

This is the logical domain for the creating and running of actors. The system defines a *namespace* for actors, allowing actors within that same system to communicate with each

other. The system is also responsible for creating the actors as they cannot be and should not be instantiated in isolation.

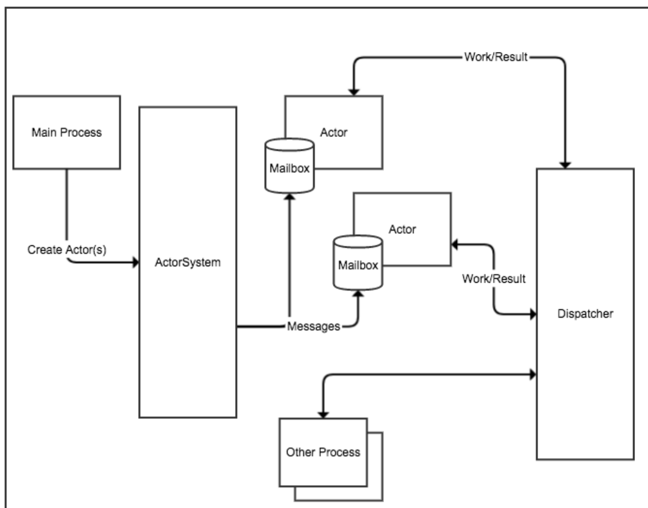
B. The Actor

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. It is to be noted that it is within the actor that the a different thread is spawned for the work to take place. It also worthy of note that the process of spawning new threads should be independent of the actor and as such interchangeable, this should eventually be determined through configuration. The actor implementation within swift will be done in a *type safe* manner so that message types are defined. This has the added advantage of predictable message handling on the side of the Actor.

C. Mailbox

The Mailbox may also be referred to as a message queue or a buffer attached to at least a single actor - this is where messages are sent to, so that the actors never directly receive messages. The *Actor System* is responsible for routing messages to the given actors mailbox and the actor then picks up the message from its attached mailbox. This is to ensure that should an actor stop working for any reason such as entering into an exceptional state and receiving a termination message from its supervisor, messages that are sent to that actor while it is shutting down are not lost and as such *buffered* in the mailbox, waiting for the next actor to take control.

These architectural components are shown in the following figure:



Basic architecture

Our general architecture owes much to the Akka Actor implementation and affords us additional capabilities:

Actor System is the logical domain for managing actors. The system encapsulates the addressing, routing, and any defined protocols for actors to communicate. Two or more actors cannot communicate with each other if they did not belong to the same actor system. Actor systems also provide the logical scaffolding for hierarchical supervision.

Supervision Multiple supervision strategies are provided describing what actions to take to bring a system back to operational integrity in the event of an actor failing. Often described as an extraneous component to the model itself - it is an integral part of modelling robustness in the model, that we believe it should be implemented as part of the basic platform.

Messaging The messaging stack includes dispatching, routing, mailbox management, and message recovery. It provides a *dead letter box* for actors that *crashed* or exited with an exceptional state. The dead letter box receives messages in *escrow* until another actor can be spawned to take the place of the dead one.

Routing Provides a means to define a routing strategy. So that given a number of actors performing the same task, which messages should go to which actor and at what frequency.

IV. Implementation

The components defined above interact with each other in very discrete manner so that Mailboxes, Actors, and Actor Systems can be created with significant independence. An actor should only be created within a systems context and as such the actor should not be constructed outside of its system. This restriction is to ensure that the actor has all the necessary properties to be able to communicate with other actors within the given system.

As this work is built upon the Swift programming language, the *Grand Central Dispatch* library [10] is utilised for the MacOS implementation. An alternative implementation is provided for the open source (and platform general) version of the language. We will concentrate on the MacOS implementation for the description included in this paper.

The implementation consists of the following constructs which map directly onto the features mentioned in the previous section. As can be seen we make use of the feature of the language to fully describe these aspects of the model.

A. ActorSystem

An extract from the *interface* is shown below. As can be deduced from the protocol defined above the system protocol is responsible for keeping track of all actors created within it, it also keeps track of the mailboxes assigned to actors. once actors have been removed from that system, the system also keeps track of mailboxes that can be reassigned to actors that are created to replace removed ones.

B. Actor

The actor has methods for “telling”, or sending messages to other actors. The method takes a message and returns nothing.

This is the component of the actor that does the work. The processor is invoked on a separate thread using the *Dispatch Framework* [10]. The actor will continuously *poll* the mailbox attached to it to ascertain whether it has messages or not. If it does it invokes the defined processor on a separate thread with the message it has just received from its mailbox.

C. Mailbox

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

V. Example usage (Pi)

We revisit the Gregory-Leibniz series [9] mentioned earlier, where the result of a series of fractions are alternatively summed up to obtain the final result. This work can be broken down into smaller pieces so that the combining operator is the addition. And as addition is associative, it lends itself to a possibility of the work being completed at different times and returned as and when values become available. To this end the different parts can be distributed among concurrently running processes and finally aggregated for the final result, as illustrated in the code below Above is a sequential calculation of pi using the Gregory-Liebniz series; our actor based version is the same but distributed across a number of threads. The result is code that is easily read and understood but has the efficiency and flexibility of a concurrent program.

VI. Related work

While our work draws principally on the Actor model there are also other programming approaches we draw upon to formulate our architecture. We will briefly describe those in this section.

A. Concurrent ML

Is an extension, which adds concurrency capabilities to Standard ML of New Jersey - a statically typed programming language with an extensible type system, which supports both imperative and functional styles of programming. CML is described as a higher-order (sometimes referred to as functional) concurrent language [11] designed for high performance and concurrent programming. It extends Standard ML with synchronous message passing, and similarly to CSP described above, over typed channels [12].

B. Occam 2

Occam is an imperative procedural language, implemented as the native programming language from the Occam model. This model also formed the bases for the hardware chip — the *INMOS transputer microprocessor* [13]. It is one of several parallel programming language developed based on Hoare's CSP[8]. Although the language is a high level one, it can be viewed as an assembly language for the transputer [13]. The transputer was built with 4 serial bi-directional links to other transputers to provide message passing capabilities among other transputers. Concurrency in Occam is achieved by message passing along point-to-point channels, that is, the source and destination of a channel must be on the same concurrent process. This notation takes its exact meaning from Hoare's CSP [8]. The "?" is requesting an input from the channel to be stored in the VARIABLE whereas "!" is sending a message over the channel and the message is the value stored in VARIABLE. Occam is strongly typed and as such the channels over which messages are passed need to be typed as well, although the type can be *ANY*, meaning that the channel can allow any type of data to be transmitted over it. An inherent limitation in Occam's data structures is that the only complex data type available is *ARRAY*.

C. 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. This 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 is a lightweight virtual machine called a process [14]. Processes communicate with each other through message passing so that a simple process written to communicate will look like the following *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.

D. Pony

Pony is an object-oriented, actor-model, capabilities-secure programming language [1]. 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[1]. 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.

E. The Akka Library

Earlier versions of Scala had natively implemented actors. Actors were 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 there is the Akka Library.

Akka is developed and maintained by Typesafe [15] and when included in an application, concurrency can be achieved. Actors are defined as classes that include or *extend* the *Actor* trait. This trait enforces the definition of at least a receive

function. In the trait *receive* is defined as a partial function with takes another function and returns a unit.

The function it expects 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.

At the heart of the Akka Actor implementation is the Java concurrency library *java.util.concurrent* [16]. 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.

F. Kotlin coroutines

Kotlin coroutines can be described as suspendable operations that can be resumed at a later time and potentially on a different thread of execution. We believe that this model can similarly be implemented in this language perhaps leveraging *coroutines*.

VII. Conclusions and Future work

In this article we have briefly described the architecture and implementation of a prototype ACTOR model implementation for the Swift programming language. The prototype can run actor based programs and is performant with manually written concurrent code. We intend to enhance our system by incorporating configuration mechanisms into the architecture to allow for different actor systems to interoperate. We also plan to add a remoting feature to allow actors to communicate across platforms and virtual environments. This will enhance the flexibility and performance of our actors.

We also plan to add an event bus, further scheduling facilities, logging, and more monitoring capabilities. From a professional usage viewpoint we need to determine a *Quality of Service* guarantee strategy, but we intend that this process can be determined after more extensive testing.

References

[1] S. Clebsch. (2015, 01) The pony programming language. The Pony Developers. [Online]. Available: http://www.ponylang.org/

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/2384716.2384720

[3] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 2nd ed. Artima Inc, 2011.

[4] R. Hickey, "The clojure programming language," in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS '08. New York, NY, USA: ACM, 2008, pp. 1:1–1:1. [Online]. Available: http://doi.acm.org/10.1145/1408681.1408682

[5] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804

[6] J. Reppy, *Concurrent Programming in ML*. Cambridge University Press, 2007. [Online]. Available: https://books.google.co.uk/books?id=V_0CCK8wcJUC

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

[8] C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[9] J. Arndt and C. Haenel, *Pi-unleashed*. Springer Science & Business Media, 2001.

[10] A. Inc. (2016) Dispatch - api reference. [Online]. Available: https://developer.apple.com/reference/dispatch

[11] J. H. Reppy, "Cml: A higher concurrent language," *SIGPLAN Not.*, vol. 26, no. 6, pp. 293–305, May 1991. [Online]. Available: http://doi.acm.org/10.1145/113446.113470

[12] J. Reppy, C. V. Russo, and Y. Xiao, "Parallel concurrent ml," *SIGPLAN Not.*, vol. 44, no. 9, pp. 257–268, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1631687.1596588

[13] D. C. Hyde, "Introduction to the programming language occam," 1995.

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

[15] Typesafe. (2014, Feb.) Build powerful concurrent & distributed applications more easily. [Online; accessed 05 Feb, 2014]. [Online]. Available: http://www.akka.io/

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