Seminar Advanced Computer Architecture
Actors

Paul Preissner
Technische Universität München

06.07.2017

Abstract

The actor model is a programming paradigm developed specifically with concurrent and distributed computing in mind. It defines a system of self-contained actors that only communicate with each other through dedicated messages and adhere to the basic semantic properties of encapsulation, fairness and location transparency. This paper will elaborate on the history, fundamental concepts and aforementioned semantic properties of the model and discuss the issues the reality of implementation entails as well as current usage of the model in commercial projects.

1 Introduction

Ever since the introduction of multi-processor systems in the 1970s [S] and even more so the introduction of multi-core processors in the early 2000s, one of the main topics in computer science has been the question of how these systems can be programmed, used and exploited to utilize them to full capacity. Now quite naturally the usage and utilization of any computer and particularly how easy it is for the programmer to do it is very much dependent on the programming language employed in a program. Over the years, most programming languages have either been adapted to enable concurrent computing in at least a basic form or efforts have been made to enable concurrent computing through additional libraries and frameworks. But since "[c]oncurrency can be solved by a good programmer in many languages, [...] it's a tough problem to solve" as former Twitter engineer Alex Payne describes [S, 5-15], it would seem counter-intuitive to use a programming language defined at a time when sequential execution was the status quo. As such in the 1970s research first delved into the task of defining languages or rather programming paradigms that are meant to primarily deal with concurrent computing tasks. One of these paradigms is the so-called actor model, as message-passing based paradigm for concurrent and distributed computing, and in that the underlying topic of this paper. To latch onto Payne again, he continues that the Actor model is "commonly used to solve concurrency problems, and it makes that problem a lot easier to solve" [S, 5-15].

Subsequently, this paper will first take a brief look at the history and a longer look at the fundamentals and basic semantics of the Actor model, followed by a discussion of the reported and expected issues the model brings with it in an actual implementation and concluded by a rundown of its most promising current usage scenarios. In this effort it is based primarily on the work of Gul Agha on the topic since the 1980s as well as his and his colleagues research in recent years.

2 Actor model history

Ideas vaguely resembling parts of the Actor model can be found in early computational models like Petri nets, Lambda Calculus or Smalltalk in that they are influenced by the message passing concept as well. According to Carl Hewitt, some of these as well as physics actually influenced him in his work on the initial definition of Actors. In the following few years, Hewitt and his colleagues, notably Irene Greif and Henry Baker, developed operational semantics and axioms for the model. In a debate between Hewitt and Edsger Dijkstra in the late 1970s the topic of unbounded nondeterminism came up, which describes the property that the delay for servicing requests can be unbounded due to contention for shared resources yet the model guarantees that the requests will eventually be serviced. Dijkstra argued that according to his model of concurrency, this property could not be fulfilled by Actors. In 1981 however, William Clinger using domain theory proved that the Actor model does indeed fulfill this property. Another milestone was Gul Agha's work on Actors in 1985 forward, completing the Actor model theory as a transition-based semantic model which still holds as the basis for most if not all discussions and implementations regarding Actors. Since then Agha with his colleagues and the groups at the California Institute of Technology, Massachusetts Institute of Technology and other international universities have researched further into the theoretical application of the Actor model in modern systems, while the MIT Message Passing Semantics Group as well as commercial projects have done a lot of practical implementation work. After years of being regarded as sort of a niche topic, the latter especially sparked up again in recent time as industries recognized the relevance of the Actor model for concurrent computing in embedded and distributed systems, especially in regards to automation as well as the establishment of the internet as a core technology of modern society and industry. [SOURCE THIS STUFF.]

3 Fundamentals & semantics of the Actor model

3.1 Fundamental concept

Very briefly summed up, in the Actor model everything is an actor. What this means is that any single computational unit is or rather can be an Actor, regardless of how complex the computations within that unit may be. As mentioned before, the model is based on the message-passing paradigm, which sees communication as the act of exchanging messages containing data and directives between whatever is defined as a computational unit. The idea of the Actor model is to encapsulate computation into actors and decouple it from the communication, which is to only be conducted via messages. As Agha puts it the model furthers one key advantage of object-oriented programming, namely the separation of an object's interface from its representation, so that now there is also a separation of control from the computational logic [SOURCE 0]. This allows Actor-based programs to be made up of "self-contained, autonomous, interactive, asynchronously operating components" [SOURCE 0] which makes them ideal for inherently nondeterministic systems such as distributed or mobile networks.

3.1.1 Actor

As per its definition, a faithfully implemented actor can only send messages to actors it already knows. This means that it is not allowed to guess or construct names or adresses of other actors [SOURCE?]. Construction of a recipient's identifier would only be allowed if all properties and valid value ranges of the identifier were known. If this restriction is not honored, there may be unanticipated consequences and issues, as will be outlined in the later chapter on the issues of implementation. Furthermore, an actor by definition when receiving a message may proceed with any and all of three possible actions given it is idle and ready to compute:

- The actor may send messages to other actors, for example as a reply to the original sender or as subsequent communication to other actors,
- create new actors for arbitrary purpose, for example as workforce for a certain parallel task,
- update its local state, in that it may do arbitrary computations within itself.

[SOURCE] The third item leads to another fundamental restriction of the Actor model, which is that an actor can only influence its own local state [SOURCE]. It is not allowed to directly change the state of other actors in any way, such changes have to be requested through messages. This as well presents a common pitfall where a faithful implementation tends to prove inefficient and is willingly omitted in favor of better performance. This, however, also brings potentially hazardous consequences. More on this is also to be found in chapter 4.

3.2 Semantics

The Actor model as per its definition has a few major semantic properties that set the basis for most of its further powers and constraints. These are "encapsulation and atomic execution of methods (where a method represents computation in response to a message), fairness, and location transparency"

[SOURCE 0-4], which if enforced already almost guarantee correct execution of the model, if not, however, lead to increased work for the programmer as the task of checking for faithful implementation falls onto them.

3.2.1 Encapsulation and atomic execution

Encapsulation essentially describes the previously mentioned property that no actor can directly influence the state of another actor but instead every actor is its own unit and may only change upon receiving a message. More precisely, ”no two actors share state” [SOURCE 0-4], subsequently describing the natural consequence of objects changing state atomically. This means that an actor upon receiving a message will compute based on the behavior defined within and will not branch into different computations if it receives another message while busy. Faithfully the actor will buffer the pending message, finish its current computation then begin computation based on the pending message, effectively completing an atomic step. While in theory for the sake of performance and perhaps also intuitively it would make sense to interrupt current computation and possibly switch to a different task the moment a message arrives or start work in parallel, this would violate the semantic definition of the Actor model, introduce hazards and compound modeling and checking of the program.

3.2.2 Fairness

In Karmani's paper, fairness is the property that ”every actor makes progress if it has some computation to do, and [...] every message is eventually delivered [...] unless the destination actor is permanently disabled” [SOURCE 0-5]. Due to the previously explained atomicity, an actor will always compute something if it received a message with instructions. This topic is tricky though, as this notion of fairness assumes that the underlying systems and schedulers are fair as well. Fairness in the actor model means that every message is eventually delivered, however this can only work if the underlying systems guarantee this as well. Given that fairness can happen as defined, one can reason more easily about the liveness properties of a program [SOURCE 0-5], as this property will then guarantee that in a composition of actor systems, no busy system can inhibit the progress of another system, which is immensely helpful for the execution of programs as a system cannot stall the entire program, in theory at least.

3.2.3 Location transparency

Previously there were already mentions of ”identifiers” for actors. One more key property of the model is that the actual location of an actor is independent of its identifier. This means that a program can be constructed without the need to know the physical topology of the executing computer system. Of course there should still be concern about latency in distributed systems for example, but most of this balancing can be offloaded to a clever scheduler, which leads to another beneficial consequence of this location transparency, which is mobility. As Karmani puts it ”Mobility is defined as the ability of a computation to migrate across different nodes" [SOURCE 0-5], meaning single actors can in theory be shifted across nodes without influencing the results of the entire program and even without need for the program to ever notice the migration. This is a key ingredient in enabling effective runtime load-balancing and reconfiguration on large scale systems, fault-tolerance in case of partial network failure given backup links and as a result improved scalability. Scalability is still one of the primary question marks for many concurrent computing models and languages as especially distributed systems keep growing both in number of computing units as well as physical distance covered, but even locally computers are growing in terms of computing units, co-processors and sensors that need to be integrated into a single composition. The two main advantages Karmani sees in mobility are the ability to balance dynamic and irregular applications at runtime, thus optimizing performance, and the ability to optimize computation for energy efficiency by balancing the program on the ideal combination of computing unit count and energy consumption per unit. However, great care needs to be taken when implementing location transparency and in consequence mobility in a real Actor framework as attempted optimizations or shortcuts can quickly prevent full or even partial migration. More on this in chapter 4 as well.

3.3 Means of synchronization and abstraction

3.3.1 Sync: a

buh.

3.3.2 Abst: a

gok.

4 Reality and issues of implementation

- in reality, some aspects aren't implemented faithfully (for efficiency, complexity)
- concerns about scalability & performance

4.1 Tools and languages

Blob.

4.2 Peculiarities in implementation

Issues.

5 Current usage of actor systems

Focus: current usage - embedded systems, cloud/distributed computing and microservices

6 Versus other models of concurrency

Expansion: comparison to other models/paradigms of concurrent computation

7 Summary

Summary.