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 Synchronization

In the context of concurrent computation, synchronization is an important aspect in the communication between workers, for the exchange of data and to ensure correctness along different phases of an algorithm or program. In the "traditional" programming models this is usually achieved with implicit or explicit procedures that are called in the affected worker, halt execution or similar in the requesting worker and then let the workers continue once the necessary data is set. In most implementations of the Actor model the idea is the same on a high level, but as per the encapsulation property has to be handled entirelyrough messages. Karmani intuitively calls this "Remote Procedure Call (RPC)-like messaging" [SOURCE 0-6] as it emulates the functionality of such a remote procedure call. The way it works is that a reque thst is sent out by an actor A which then switches to waiting for replies. If it receives a reply by any actor B that matches the request, A can proceed with its computation. If not, B's message has to be deferred for later processing in some way until the expected reply arrives [SOURCE 0-6]. Exactly how this is implemented is down to the individual model instance and for the sake of usability it is preferable if the actor library or language in question provides a high-level abstraction. RPC-like messages offer themselves as a straight forward solution for achieving ordered processing of a message sequence or simply when all next actions of the requestor depend on the reply. Another approach are so-called local synchronization constraints. The idea here is to define a set of constraints on an actor the specifies the order it should keep and process messages in depending on its current state and the type and content of the messages, for example by

inspecting the content and buffering messages that are out of order [SOURCE 5-34]. Karmani suggests that these approaches can be used to construct patterns such as pipelining and divide-and-conquer to map actors to a more complex task distribution in a modular fashion [SOURCE 0-6]. Alternatively, specialized constructs called Synchronizers could be thrown into the mix and act as a sort of pre-processing of messages that ensures the correct order, effectively shifting ordering and synchronization outside of the individual actors.

3.3.2 Abstraction

Patterns may also be used in a different context, abstraction. As Karmani points out, ideally a programmer can utilize a language or library in a fashion similar to the conceptual level of the problem to be solved and it may be counterintuitive to work with actors as single units. Instead one would prefer to have abstractions that compose actors in a way that aligns them with the concept of the program. [SOURCE 0-7] Patterns can be employed to structure communication, in that groups of actors are assigned properties by which they can be identified or discarded as recipients for certain types of messages. For example actors can specify that they are only interested in messages containing specific data and belong to a certain group. Likewise a sender might specify properties that the recipients of its message need to satisfy. A message can then be sent out to a satisfying group without the need to perform a lot of individual checks or synchronization. Other abstractions and also sychronization techniques are certainly possible but might be more specifically tailored for the respective problem that is to be solved or the primary application field of the library or language.

4 Reality and issues of implementation

Looking at the definition of the Actor model one can see that while the different semantic properties and constraints do make sense to ensure proper and safe execution of a program within the confines of the model, the reality of implementation is different. Not all properties can be implemented faithfully while still performing well, or at least might require major optimization work to reach viable levels of efficiency. This will inevitably lead many programmers attempting to implement an Actor language or library to take shortcuts to achieve better performance or reduce work effort. The issue then is that shortcuts or supposed optimizations will often violate the properties and constraints leading to flawed execution and unexpected bugs. Karmani notes that "a faithful but naive implementation of the Actor model can be highly inefficient" [SOURCE 0-8] and currently only usable for coarse-grained concurrency [SOURCE 1-7] but this "may be adressed by compilation and runtime techniques" [SOURCE 0-8]. This chapter will outline a few potential pitfalls and potential solutions.

4.0.1 Message latency

One problem that is basically natural to distributed systems (yet also local small scale systems to a lesser degree) is latency. The further a signal or message for that matter has to travel between actors, the higher the latency between send and receive is. On a conceptual level this is a simple issue to fix by overlapping communication and computation. This way computation can to a degree mask the delay introduced by messaging, limited to the maximum time the computation takes for given possible inputs. Of course this is not a full fix though and can still be outweighed by a disproportional surplus of messages. Thus a program also needs to be engineered with caution as to not overburden it with communication. Another role in achieving good overlap falls to decomposition of the program into suitable actors and the placement of the respective actors, which can be done using the mobility property of the Actor model. [SOURCE 0-8] Another possibly more crude aspect that can contribute to lower latencies is improved network interconnects, be it connections between distributed clients or data buses in embedded systems. While this is not a guaranteed fix either and might seem more as a cure for the symptoms of unoptimized actor systems rather than for the cause, it is a factor to consider.

4.0.2 Naive send vs channels

It was previously established that actors may only send messages to actors they know. Some languages and libraries modify this idea towards the concept of channels, which specify in so-called stateful channel contracts "a protocol that governs the communication between two end-points (actors) of the channel" [SOURCE 0-8]. This way the type of messages allowed as well as the necessary order can be specified in a more intuitive way as opposed to individual checks of messages in each actor and additional synchronization efforts. [SOURCE 0-8]

4.0.3 Garbage collection

Another topic Karmani briefly mentions is garbage collection, which as of yet is an unresolved issue in distributed systems as it is not yet solved how to efficiently check for reachability across the entire system. [SOURCE 0-8]

4.0.4 Thread overhead and context switches

A basic Actor language or library might opt for mapping an actor to a classic thread for simplicity. The issue this presents is overhead due to context switching and thread creation. Every time the program execution switches to a different actor and thus thread it has to perform a complete context switch including saving its entire stack, counters and registers. This problem becomes increasingly prevalent the more actors a program uses. Karmani presents one possible solution in continuations based actors which don't perform a full fat context switch and significantly reduce overhead for switching and creation. In his testing, going from traditional threads to continuations in the Actor Foundry

language reduced runtime of a thread switching benchmark by over 60% from 695s to 267s. [SOURCE 1-8]

4.0.5 Deep copies vs referencing

By definition, an actor may not change the state of another actor directly. In a naive implementation on a shared memory system this means that any data exchange would need to be happen by copying the data into a message, which is quite slow in contrast to call-by-reference. Sending a reference however carries that risk of direct changes happening and violating the model. Karmani's optimization efforts suggest carefully allowing immutable types to be sent by reference. According to his numbers in the aforementioned benchmark, sending the main message content type by reference reduced runtime by another 84%. [SOURCE 1-9] Of course the speedup achieved this way heavily depends on which message content types are most common in a program as well as the scale of the system, with larger systems inherently presenting higher risks for deep copy slowdown.

4.0.6 Scheduling

Once more the definition of the Actor model specifies a property that depends heavily on implementation. The model claims fairness in that every actor will make progress if it has pending messages. This, however, has to be ensured by the scheduler. This is not inherently an issue, but can be if the scheduler underneath the used language or library does not comply. What Karmani did for his testing was to write a custom scheduler which monitors the worker threads and spawns a new thread when no progress is made yet actors are queuing to be scheduled. While not highly sophisticated, his method when tweaked performs almost identical to the default JVM scheduler but does guarantee fairness. [SOURCE 1-9]

4.0.7 Safe messaging

In the discussion of his paper, Karmani mentions safe messaging as a remaining major topic for optimization. This goes back to deep copying versus referencing message content. It remains an active research topics how it could be reliably and efficiently determined which types may be safely sent by reference.

4.1 Tools and languages

Compared to "traditional" languages, tool support for actor systems is still rather basic. There are some plug-ins for Erlang and Scala for Eclipse for example and some testing tools such as QuickCheck for Erlang and Basset or JCute for Java actors exist, but more widespread, sophisticated and perhaps centralized support is still an active development topic. [SOURCE 0-10] Many languages are actually supported in that actor libraries exist for use with them, and plenty languages have been specifically designed around the actor model,

and many of these are still actively supported. Some notable actor languages include Erlang, Scala, SALSA and the Ptolemy project along with at least two dozen other languages. Libraries exist for basically all common major languages like Java, C, C++, Python, Haskell, .NET, C#, Ruby, JavaScript, LabVIEW and more. One issue that can be seen with some of these libraries, however, is that while they are still actively updated, they tend to have lackluster documentation, small communities and little developer support. It is noteworthy though that these languages and libraries cover wide ground in terms of the system structures they support, be it distributed systems, embedded systems, small scale or large scale. [SOURCE 0W]

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.