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 and multi-core processors in the early 2000s, one of the main topics in computer science has been how these systems can be utilized to their full capacity. Naturally the utilization of any system and how easy it is for the programmer to do is very dependent on the programming language employed. Over the years most programming languages have been adapted to enable concurrent computing in some form, either natively or through libraries. But since "[c]oncurrency can be solved by a good programmer in many languages, [...] but it's a tough problem to solve" as former Twitter engineer Alex Payne describes [S, 5-15], it seems counter-intuitive to use a language originally defined for sequential execution. As such in the 1970s research first delved into defining languages and paradigms meant to primarily deal with concurrent computing. One of these paradigms is the actor model, a message-passing based paradigm and in that the underlying topic of this paper. To latch onto Payne again, 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 look at the history, fundamentals and basic semantics of the Actor model, followed by a discussion of some known issues the model entails in implementation, concluded by a rundown of promising current usage scenarios and brief comparison to other models.

2 Actor model history

Similarities to early computational models such as Petri nets and Lambda Calculus can be found in the actor model's message passing concepts. According to Carl Hewitt, some of those as well as physics influenced his work on the definition of actors in the 1970s. 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 question arose whether actors satisfy unbounded nondeterminism, which describes 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. In 1981 William Clinger using domain theory proved the Actor model does indeed fulfill this property. Another milestone was Gul Agha's work on Actors from 1984 forward, formalizing the Actor model as a transition-based semantic model. Since then Agha and colleagues and groups at CalTech, MIT and other international universities have researched further into the theoretical application in modern systems while the MIT Message Passing Semantics Group and commercial projects have done a lot of practical implementation work. The model sparked up again in the past decade as industries recognized the relevance of the model for embedded and distributed systems. [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. 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 exchange of data/directive messages between whatever is defined as a computational unit. Computation is encapsulated into actors and while communication is only conducted via messages. As Agha puts it, the model furthers one key advantage of object-oriented programming, the separation of an object's interface from its representation, and expands it by 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. 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 are known.

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 action leads to another fundamental restriction of the Actor model, 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 presents a common pitfall where a faithful implementation tends to prove inefficient and is willingly omitted in favor of better performance, introducing hazards in execution.

3.1.2 Sample actor program

To illustrate actors, the C++ Actor Framework [SOURCE CAF] was used to implement a small Ping Pong program as in listing 1. An Actor A sends a string "Ping" to an Actor B. Actor B replies to any incoming messages matching to "Ping" with a reply "Pong". Actor A reacts to any incoming "Pong" with yet another "Ping". The output of this program is simply a continuous back and forth of "Ping" and "Pong". It does show, however, that using actors in programs proves fairly easily given a properly implemented library. Note that no thread management or balancing has to be done by the high level programmer. It is only necessary to specify the behaviors for actors and kickstart a first call on at least one actor. The rest is done almost entirely by the library and this small sample program is scheduled by CAF to utilize all four threads of the executing Intel Core i7-7500U to at least 75% utilization.

3.2 Semantics

The Actor model by definition has a few major semantic properties. 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 almost guarantee correct execution of the model, if not, however, put the task of checking for faithful implementation onto the programmer.

3.2.1 Encapsulation and atomic execution

Encapsulation describes the previously mentioned property that no actor can directly influence the state of another actor. More precisely, "no two actors share state" [SOURCE 0-4] and on a conceptual level change state atomically.

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 continue with the pending message, effectively completing an atomic step. While in theory for the sake of performance it makes sense to interrupt current computation and switch to a different task the moment a message arrives or start work in parallel, this violates the semantic definition of the Actor model, introduces hazards and compounds 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. This notion of fairness assumes that the underlying scheduler is fair as well. Given that, one can reason more easily about the liveness properties of a program [SOURCE 0-5] as this property guarantees that in a composition of actor systems, no busy system can inhibit the progress of another system thus stall the entire program.

3.2.3 Location transparency

One more property of the model is that the actual location of an actor is independent of its identifier. A program can be constructed without the need to know the physical topology of the executing system. This facilitates another property, mobility. As Karmani puts it, "Mobility is defined as the ability of a computation to migrate across different nodes" [SOURCE 0-5], meaning actors can be shifted across nodes without affecting the correctness of execution. This is key for effective runtime load-balancing of irregular and dynamic applications, energy efficiency optimization, fault-tolerance in case of partial network failure and improved scalability. Scalability is still one of the primary question marks for many concurrent computing models and languages as distributed systems keep growing both in number of computing units and physical distance covered, and single computers in terms of cores, co-processors and sensors that are integrated into a single composition.

3.3 Means of synchronization and abstraction

3.3.1 Synchronization

Synchronization in the actor model cannot be achieved with traditional means such as explicit functions calls due to the encapsulation property. Instead this has to be handled through messages. Karmani describes a solution based on the same principle though, "Remote Procedure Call (RPC)-like messaging" [SOURCE 0-6]. A request is sent out by an actor A which then waits. If it receives a matching reply by any actor B, A proceeds. If not, B's message is deferred until the expected reply arrives [SOURCE 0-6]. Specific implementation is down to the individual library or language. RPC-like messages offer themselves as a solution for ordered processing of messages or when all next actions of the requestor depend on the reply. Another approach are so-called local synchronization constraints. A set of constraints is defined on an actor that specifies the order it should keep and process messages in, depending on its state and the message type and content. [SOURCE 5-34] 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. Karmani suggests that these approaches can be used to construct patterns like pipelining and divide-and-conquer to map actors to a more complex task distribution in a modular fashion [SOURCE 0-6].

3.3.2 Abstraction

Patterns may also be used in a different fashion for 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 at hand. Instead of treating actors as single actors one would prefer to have abstractions that compose actors in a useful way. [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 messages. For example actors can specify their group and that they are only interested in specific message data. A sender can specify properties the recipients of its message need to satisfy. A message can then be sent out to a satisfying group without performing many individual checks. Other abstractions and sychronization techniques are certainly possible but might be more specifically tailored for the respective problem or 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

5.0.1 Actors in distributed systems

One relevant field for actors are distributed systems. There are multiple levels at which the model can be applied. An intuitive option is to model clients as actors, and essentially map the physical structure of the system 1:1 or cluster several clients into an actor. Communication then map to the physical links between clients. This may not be exactly efficient or suitable to the application though. The more useful mapping is to decoupled parts of the program. For example an actor may be mapped to a user of a service, or to a certain part of a service itself. This is the case for commercial (as well as free and open source) systems such as the Facebook chat backend [SOURCE FB], Twitter message queues [SOURCE TW], even the game servers for the video game Halo 4 [SOURCE H4] or the web application framework Lift [SOURCE 5-14]. Other projects exist in the high performance computing field, demonstrated for example by the ActorX10 group of the Invasic project at TUM which uses a fusion of X10 and the actor model and illustrates how actors may be used to build highly scalable programs for large computing clusters while also helping developers by simplifying communication and separating control flow from computation [SOURCE 6-4]. Erlang and Scala appear to be the most popular languages as of this date due to their developer support and in-depth documentation. A similar potential lies in so-called microservices, essentially a type of service-oriented architecture where collections of independent services that are composed into a whole application. The idea is to achieve better scalability, decentralized control and fault-tolerance, which actors are an almost perfect fit for [SOURCE MS].

5.0.2 Actors in embedded systems

Just as the actor model is suitable to map clients and connections on a large scale, it is also suitable to map programs, algorithms and computational units on a very small scale in embedded systems. In their research into actor-oriented design of embedded systems in 2002, Edward A. Lee et al. argue that this approach is "particularly effective for system-level design" [SOURCE 2-12]. Individual components of the system are modeled as actors, such as parts of an algorithm, especially when they are reusable for different computations as well as to enable better reasoning over execution within the system. The actor model offers them a clear method for abstraction and definition of interfaces, object representations and connections, however in their specific case quite constrained to only enable very specific communication and interaction. Lee et al. state that "[t]he primary benefit of actor-oriented design is the possibility of succinctly capturing the requirements of an embedded system by the modeling properties of a model of computation" [SOURCE 2-23], thus specifically satisfying the requirements of model-based design. Contrary to this approach of using the actor model in the design stage, some efforts also use the model directly in the implementation of programs on embedded systems. The previously mentioned ActorX10 library illustrates this by implementing a object detection module for computer vision systems. They utilize a SIFT-style algorithm that features multiple sequential stages. Each stage is modeled as an actor which receives data messages from the previous stage actor and sends its result message to the next. Now this structure can be exploited for parallelism by pipelining the computation. Once a stage has finished computation from one input, it can immediately take the next input data and work on the next object. This way every stage can have essentially 100% uptime and, after a short startup time until each stage has data to work with, can output a result each time the last stage finishes. [SOURCE 6-4] The advantage of this is that parallel execution works on a modular level instead of running the entire sequence in multiple parallel instances. This could allow even further optimization, for example in hardware through specialized ASICs for certain stages. Omer Kilic of the Erlang Solutions group also sees the actor model very suited for current challenges in computing technology such as the creation of complex SoCs, IoT devices and heterogenous architectures [SOURCE 3-5].

6 Versus other models of concurrency

6.0.1 vs Petri nets

Petri nets on a basic conceptual level are mathematical graph-based models with two types of nodes, places and transitions. Places are equal to conditions of any form while transitions are possible events. Flow within the model is represented by arcs which can point from either a place or a transition to the respective other. Arcs may not be put between nodes of the same type. This way the net essentially describes a complex system of pre- and postconditions for possible events. Every such place/condition can have an arbitrary number of tokens which can each trigger a transition and are then "transferred" to the postcondition of that event. [SOURCE PN] This can be exploited for concurrency by placing multiple tokens wherever necessary in the net. The largest difference to the actor model appears to be that actors are modeled as the (abstract) computing units themselves with communication between each which can lead to state changes while a petri net describes the possible changes and events mostly independent of the representation of computation.

6.0.2 vs process calculi

Process calculi are not one model specifically but rather a collection of similar formal models. Their most basic definitions all include a few key properties. Processes communicate/interact through message-passing, very similar to the actor model, however primarily anonymously via channels, unlike the pure actor model. Processes are modeled as a composition of primitives and operators, unlike the actor model which does not provide tools to specifically model the actions within an actor per se. These operators are subject to algebraic laws to facilitate reasoning about the system. [SOURCE calculi] The similarities between the two models do not come as a surprise considering Carl Hewitt named process calculi as one influence on his work. He also noted, however, that his actor axioms are inspired by physics, meanwhile process calculi are in essence algebraic.

6.0.3 vs input/output automaton

I/O automata approach the issue of concurrency at an even more modular level. They can be used to model single components of almost any kind, be it computational units, data or communication methods. An automaton is essentially a state machine with inputs, outputs and some sort of internal action. Only the in- and outputs may be used to communicate with other automatons, internals are hidden. [SOURCE I/OA] In this way there is some resemblance to the encapsulation in the actor model, however the actor model does already specify communication and basic actor structure. The automata could in theory be used to model an actor system, however with more effort.

7 Summary

Summary.

References