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 fundamental concepts and aforementioned semantics of the model, discuss the issues of implementation, current practical usage of the model and a brief comparison with other concurrency models.

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, depending on the programming language employed. Most programming languages have been adapted to enable concurrent computing either natively or through libraries. 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 [2, p. 15], it seems counter-intuitive to use a language originally defined for sequential execution. Therefore 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, initially developed by Carl Hewitt and his colleagues, formalized by Gul Agha in 1985 and researched by multiple groups and universities since then [4]. The model is "commonly used to solve concurrency problems, and it makes that problem a lot easier to solve" [2, p. 15]. Subsequently, this paper will take a look at the fundamentals and basic semantics of the actor model, followed by a discussion of some issues the model entails in implementation and a look at the potential it offers nowadays, concluded by a brief comparison to other models.

2 Fundamentals & semantics of the Actor model

2.1 Fundamental concept

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 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 [4]. This allows actor-based programs to be made up of "self-contained, autonomous, interactive, asynchronously operating components" [4, p. 1] which makes them ideal for inherently nondeterministic systems such as distributed or mobile networks.

2.1.1 Actor

According to 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 addresses of other actors [4]. 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 may proceed with any and all of three possible actions when receiving a message, given it is idle and ready to compute (as seen in 1):

- 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.

The third action leads to another fundamental restriction of the actor model: an actor can only influence its own local state . It is not allowed to directly change the state of other actors in any way, such changes have to be requested through messages[4]. This presents a common pitfall where a faithful implementation tends to prove inefficient and is willingly omitted in favor of better performance.

2.2 Semantics

The Actor model 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" [4, p. 4] which if enforced almost guarantee correct execution of the model. If not, however, they put the task of checking for faithful implementation onto the programmer.

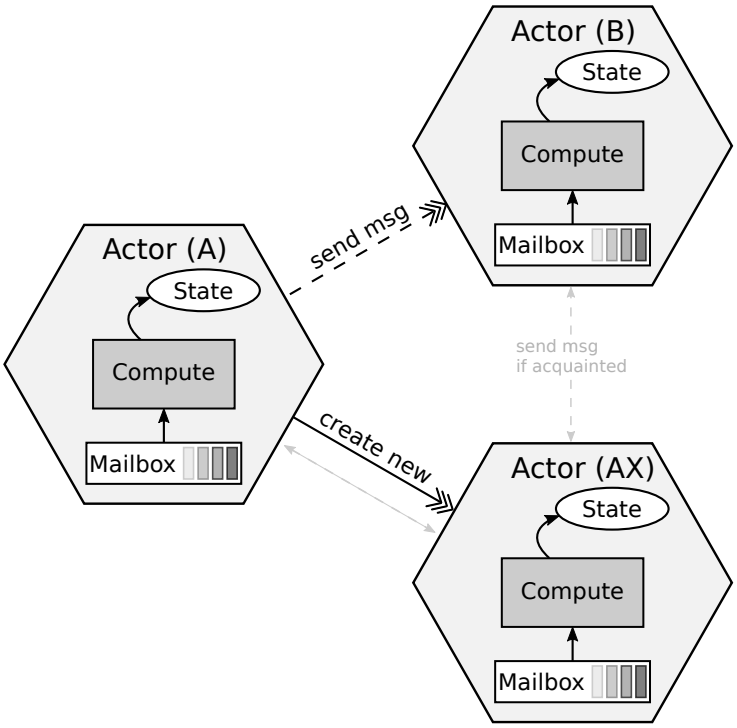

Figure 1: Any actor may send messages to other known actors, create new actors or update its own state.

2.2.1 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" [4, p. 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. Theoretically, for the sake of performance, it makes sense to interrupt current computation and switch to a different task the moment a message arrives or starts to work in parallel, however, this violates the semantic definition of the Actor model, introduces hazards and compounds modeling and checking of the program.

2.2.2 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" [4, p. 5]. Due to 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 [4, p. 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.

2.2.3 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. "Mobility is defined as the ability of a computation to migrate across different nodes" [4, p. 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.

2.3 Means of synchronization and abstraction

2.3.1 Synchronization

Synchronization in the actor model has to be handled through messages. Karmani describes a solution based on the same principle as traditional procedure calls, "Remote Procedure Call (RPC)-like messaging" [4, p. 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

[4, p. 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 over an actor that specifies the order it should keep and process messages in, depending on its state and the message type and content. [2, p. 34] Alternatively, specialized constructs called *synchronizers* can be used to act as a pre-processing of messages that ensures 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 [4, p. 6].

2.3.2 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. [4, p. 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.

3 Reality and issues of implementation

While these semantics are logical in theory, the reality of implementation is different. Not all properties can be implemented faithfully while still performing well without major optimization work. This might lead to programmers attempting to implement an actor language or library to take shortcuts to achieve better performance. But those supposed optimizations might violate the properties and constraints leading to flawed execution. Karmani notes that "a faithful but naive implementation of the Actor model can be highly inefficient" [4, p. 8] and may only be feasible for coarse-grained concurrency [5, p. 7] but this "may be addressed by compilation and runtime techniques" [4, p. 8]. This chapter will outline a few potential pitfalls and potential solutions. Note that garbage collection and safe messaging are mentioned by Karmani as yet unresolved issues[5, p. 10], but will not be discussed here due to lack of clear options for solution.

3.0.1 Message latency

Latency is natural to any system, especially distributed ones. The further a message has to travel, the higher the latency between send and receive is. Conceptually this is fixed by overlapping communication and computation. This

way computation can partially mask the latency. This can still be outweighed by a disproportional surplus of messages, therefore a program needs to be engineered as to not overburden communication. Good overlap also needs suitable decomposition of the program into actors and smart migration. [4, p. 8] Another cruder aspect that may lower latencies is improved network interconnects, be it connections between distributed clients or data buses in embedded systems.

3.0.2 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, specifically with stateful channel contracts, "a protocol that governs the communication between two end-points (actors) of the channel" [4, p. 8]. This way the allowed type and order of messages can be specified in a more intuitive way as opposed to individual checks of messages in each actor and additional synchronization. [4, p. 8]

3.0.3 Thread overhead and context switches

A basic actor language or library might opt for mapping an actor to a 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, 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 context switch and significantly reduce overhead. In his testing, going from traditional threads to continuations in ActorFoundry reduced runtime of a thread switching benchmark by over 60% from 695s to 267s. [5, p. 8]

3.0.4 Deep copies vs referencing

By definition, an actor may not change the state of another actor directly. Naively on a shared memory system any data exchange would need to happen by copying the data into a message, which is slow compared to sending a reference. Sending a reference however carries the risk of direct changes violating the model in the process. Karmani's optimization efforts suggest 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%. [5, p. 9] The speedup achieved this way depends on which message content types are most common in a program as well as the scale of the system.

3.0.5 Scheduling

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. It's not inherently

an issue but can be if the scheduler 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. [5, p. 9]

4 Current usage of actor systems

4.0.1 Actors in distributed systems

There are multiple levels at which the model can be applied in distributed systems. An intuitive option is to model clients as actors and map them 1:1 or cluster several clients into an actor. Communication then maps to the physical links between clients. This may not be efficient or suitable for the application though. A more useful mapping is to decouple and map parts of the program, such as a user of a service or a certain part of a service itself. This is the case for systems such as the *Facebook* chat backend [8], *Twitter* message queues [14], game servers for the video game *Halo 4* [17] or the web application framework *Lift* [2, p. 14]. Other projects exist in high performance computing, demonstrated for example by the *ActorX10* group of the *Invasic* project at TUM which uses a fusion of X10 and the actor model to illustrate how actors can be used to build highly scalable programs while also simplifying communication and separating control flow from computation [16, p. 4]. A similar potential lies in so-called microservices, a type of service-oriented architecture where collections of independent services are composed into a whole application. The idea is to achieve better scalability, decentralized control and fault-tolerance. [9]

4.0.2 Actors in embedded systems

The model is also suitable to map programs and algorithms on a very small scale in embedded systems. About actor-oriented design of embedded systems in 2002, Edward A. Lee et al. argue this approach is "particularly effective for system-level design" [7, p. 12]. Components of the system, such as parts of an algorithm, are modeled as actors 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. 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" [7, p. 23], specifically satisfying the requirements of model-based design. Contrary to using actors in the design stage, some efforts use the model directly in the implementation of programs. The previously mentioned ActorX10 library illustrates this by implementing an object detection module using an algorithm that features multiple stages. Each stage is modeled as an actor receiving data messages from the previous stage actor

and sending its result to the next. This structure can be exploited for parallelism by pipelining. Once a stage is finished with one input, it can immediately work on the next input data. Every stage can have almost 100% uptime and a final result is had each time the last stage finishes. [16, p. 4] 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 [6, p. 5].

5 Versus other models of concurrency

5.0.1 vs Petri nets

Petri nets on a 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 pointing from either a place or a transition to the respective other. The net 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. [10][13] This can be exploited for concurrency by placing multiple tokens 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.

5.0.2 vs process calculi

Process calculi are not one model specifically but a collection of similar formal models. Their basic definitions all include a few key properties. Processes communicate/interact through message-passing, similar to the actor model, however 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 specifically model the actions within an actor. These operators are subject to algebraic laws to facilitate reasoning about the system. [3] The partial similarities between the two models do not come as a surprise considering Carl Hewitt named process calculi as one influence on his work.

5.0.3 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. [11] 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.

6 Conclusion

To summarize, this paper offers a concise overview of actors. It explains the fundamentals of the actor model, the semantic properties encapsulation, fairness and location transparency (and with it mobility), synchronization through RPC-like communication and local constraints, as well as patterns for abstraction. It was established that while a faithful implementation of the model is inefficient, several approaches can be taken to optimize runtime behavior in latency masking, sending efficiency and parallelism overhead. Lastly, current usage of the model was discussed. It's seems very well suited for distributed computing as well as embedded systems as its structures and communication can be nicely mapped to such systems and facilitate correctness checking. Other models of concurrency share some properties with the model, yet it stands out with distinct advantages. Unfortunately, the extent of this paper does not allow deeper analysis of the model or an expanded view on its usage in modern commercial or free/open source projects as the presented overview hints at a wide range of possible applications. It is clear, however, that it has great potential for parallelizable challenges in computing and more research is needed to materialize this potential.

References

[1] Gul Agha. *Abstractions, Semantic Models and Analysis Tools for Concurrent Systems: Progress and Open Problems*, pages 3–8. Springer International Publishing, Cham, 2016.

[2] Gul Agha. Abstractions, semantic models and analysis tools for concurrent systems: Progress and open problems - sefm keynote 1 at staf 2016, 2016. http://staf2016.conf.tuwien.ac.at/wp-content/uploads/SEFM2016-GulAgha.pdf.

[3] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, May 2005.

[4] Rajesh K. Karmani and Gul Agha. Actors. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1–11. Springer, 2011.

[5] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In Ben Stephenson and Christian W. Probst, editors, *PPPJ*, pages 11–20. ACM, 2009.

[6] Omer Kilic. The actor model applied to the raspberry pi and the embedded domain - the erlang embedded project, 2013. https://www.slideshare.net/omerk/the-actor-model-applied-to-the-raspberry-pi-and-the-embedded-domain.

[7] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12:231–260, 2003.

[8] Eugene Letuchy. Facebook chat, 2008. https://www.facebook.com/note.php?note_id=14218138919.

[9] James Lewis and Martin Fowler. Microservices - a definition of this new architectural term, 2014. https://martinfowler.com/articles/microservices.html.

[10] Chris Ling. The petri net method, 2001. www.utdallas.edu/ gupta/courses/semath/petri.ppt.

[11] N.A. Lynch. *Distributed Algorithms*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 1996.

[12] Multiple. Actor model - programming with actors, 2017. https://en.wikipedia.org/wiki/Actor_model#Programming_with_Actors.

[13] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[14] Alex Payne and 'Glenn'. Engineer-to-engineer talk: How and why twitter uses scala, 2010. https://redfin.engineering/engineer-to-engineer-talk-how-and-why-twitter-uses-scala-81e036cba7d.

[15] Shangping Ren and Gul Agha. A modular approach for programming embedded systems. In Grzegorz Rozenberg and Frits W. Vaandrager, editors, *European Educational Forum: School on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 170–207. Springer, 1996.

[16] Sascha Roloff, Alexander Pöppl, Tobias Schwarzer, Stefan Wildermann, Michael Bader, Michael Glaß, Frank Hannig, and Jürgen Teich. Actorx10: An actor library for x10. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, pages 24–29, New York, NY, USA, 2016. ACM.

[17] Jan Stenberg. Abstractions, semantic models and analysis tools for concurrent systems: Progress and open problems, 2015. https://www.infoq.com/news/2015/03/halo4-actor-model.