https://alex-karaberov.medium.com/everything-you-always-wanted-to-know-about-the-actor-model-but-were-afraid-to-ask-b6eee8722953

Alexander Karaberov: Everything you wanted to know about the Actor Model but might have been afraid to ask

In this blog post, I would like to explore the fundamental aspects of the Actor Model. This technical essay aims to help you understand the key ideas behind the Actor Model, as well as its main premises as stated by Professor Carl E. Hewitt in his seminal paper. Another reference for this article is the paper Laws for Communicating Parallel Processes by Hewitt and Baker, which presents a mathematical framework and the laws of the Actor Model.

UPD: Professor Carl E. Hewitt somewhat approves this article

I wish not to inundate you with the intricacies of particular implementations, such as Erlang/OTP and Elixir processes or Akka actors. Furthermore, controversial as it may seem, I am unaware of any mainstream industrial technology or programming language that implements the ***pure*** Actor Model without some degree of deviation from the original model. Therefore, the objective of this article is to elevate the level of abstraction and present a *conceptual specification* of the Actor Model. Please be aware that, notwithstanding my efforts to maintain rigor, this article should not be considered a treatise of any kind.

And now, without further ado, let's start.

The Actor Model is a mathematical theory of computation, based on the concept of **Actors**. Actor is a fundamental unit of computation, which embodies three things:

1. Information processing (computation).
2. Storage (state).
3. Communication.

Actors are the fundamental and **only** objects in the model, consequently *everything is an actor*. There can not be a system of one actor, because an actor has to **communicate** with other actors. Actors interact with each other *only* by one actor sending another actor a **messenger** (message), which is *also an actor*. Message and messenger are completely interchangeable notions and I will use both of them.

You may have already noticed the similarities between the Actor Model and pure object-oriented programming following in the footsteps of Alan Kay, where everything is an object, messages are sent between objects, and objects have local mutable state.

Nevertheless, it is crucial to acknowledge a significant distinction: the boundary between actors is *immutable*, precluding an actor from passing a mutable reference or altering another actor's state in any way. Actor can **only** send a message.

Actors are independent entities whose model does not restrict them to a process or a thread, they have an ability to communicate across processes, pipes, machines. Therefore merely to recapitulate — OOP models behaviour and state, whereas Actor Model models (pardon a tautology) computation and is a precise, complete, formally defined theory as opposed to OOP.

The actor to which the messenger is sent is called the **target**. Thus, the only kind of *event* in this model of computation is the receipt of a messenger by a target. ***Events*** are the discrete steps in the ongoing history of an actor computation; they are the fundamental interactions of the actor theory. Every event **E** consists of the *receipt* of *a* **message(E)**, by a target (recipient) actor, called **target(E)**. Terminology is based on the paper Laws for Communicating Parallel Processes (Hewitt and Baker).

Upon the receipt of a message, a target actor reveals **behaviour** by sending messages to other actors. Actors can be created by another actor as part of the second actor's behaviour. Indeed, almost every messenger actor (message) is newly created before being sent to a target actor. For each actor *X*, we shall require that there is a unique event *Birth(X)* in which *X* first makes its appearance.

More precisely *Birth(X)* has the property if *X* is a participant in another event **E** then *Birth(X)* → ***E***, where → is a binary relation of events, which mean a partial ordering of events. Formally it will be a transitive closure of the union of the activation ordering and the arrival orderings of events in the Actor Model.

Each actor has an **address**. In various implementations an address can be a direct physical address such as a MAC address of the NIC, a memory address, or simply a process identifier (PID). Multiple actors can have the same address, and one actor can have multiple addresses. There is a *many-to-many* relationship here. Address is **not** a unique identifier of the actor. Actors don't have an identity, only addresses. So, when we step back and look at our conceptual Actor Model, we can see and use only addresses. We can not tell whether we have one actor or multiple ones even if we have one address, because it can be a proxy for the group of actors. All we can do with an address is send it a message. Address represents *capability* in the Actor Model. Mapping of addresses and actors is not part of the conceptual Actor Model although it is a feature of implementations.

Actors are allowed to send messages to themselves (recursion support), which they will receive and handle later in a future step. Also, actor *may* have a mailbox. **Mailbox** is an actor (remember: *everything is an actor*), which represents messages' destination point. Mailboxes are not required by an Actor, because if an Actor was required to have a mailbox then, the mailbox would be an Actor that is required to have its own mailbox and we end up with an infinite recursion.

There exists two **axioms of locality** in Actor model, including **Organisational** and **Operational**.

Operational:

In response to a message received, an Actor can **only**:

1. Spawn a finite number of new actors.
2. Send messages to addresses *only* in the message it has just received or in its local storage.
3. Update its local storage for the next message (designate what to do with the next message)

Organisational:

The local storage of an Actor can include addresses **only**:

1. That were provided when it was created
2. That have been received in messages
3. That are for Actors created here (paragraph 1 of Operational axiom)

Conceptually an actor processes incoming messages from his mailbox sequentially one message at a time, but physical implementations always optimise or pipeline message processing somehow. The actor model does not postulate many guarantees for messaging and the processing. Different physical implementations are free to add interesting features on top for instance messages pattern matching or selective receive in Erlang to queue unmatched messages in the mailbox for later processing and managing timeouts.

The Actor Model also supports **delegation**, because actor can send addresses of other actors in the message (in Erlang these are PIDs).

As I already mentioned before, an actor can send messages to itself (recursion) and in order to avoid deadlocks, we have a notion of the **future** in the Actor Model. The idea of a future is that you can create an actor with any result whilst it's still being computed. **Future** is a special type of the message, which represents "future" value of (potentially very long) computation or event (result of sending a message to a target). Actors can pass futures to other actors and they can send future to themselves. Yes, these **are the same futures** we now have in mainstream programming languages such as JavaScript or Scala. The notion of the future first appeared in the Actor Model in 1977 ("The incremental garbage collection of processes" by Carl Hewitt and Baker).

There is no guaranteed order for message delivering and messages can be dropped (e.g. if Actor was destroyed before sending a messenger), so we have a best-effort delivery. Messages can be persisted (paragraph 2 of Actor definition: *storage*) and can be resent. Message can be delivered **at most** once (either one or zero). Rapidly arriving messages could result in a sort of denial-of-service for

the actor, rendering the actor incapable of processing the incoming message flow. To alleviate this problem, there exists a mailbox actor receiving messengers and holding those messengers until the actor is able to process them. Messages can take arbitrarily long to eventually arrive at the mailbox of the receiver. In physical implementations of the Actor Model queuing and dequeuing of messages in a mailbox are atomic operations, so a race condition is not possible.

The fact of an actor having its own state or storage implies additional desirable requirement, notably actors should be inherently persistent unless no longer reachable. Hence an Actor System should restore Actors based on recordings that it has. We can denominate this property an “actor’s storage durability”. Durability here is a matter of degree. Lowest is to record in RAM. Next is persistent local storage (local database server, et cetera). After that comes storage on other machines which can be represented as a replicated journal (log) backed by a distributed database. An interesting physical implementation of this property would be Akka Persistence. **Akka persistence** enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash, or by a supervisor, or migrated in a cluster. The key concept behind Akka persistence is that only changes to an actor’s internal state are persisted but never its current state directly (except for optional snapshots).

There are no channels or any other kind of intermedium between actors. We send messages *directly* to the actors. Of course we can implement a channel via an actor with “put” and “get” messengers, but this is not required at all.

Computation in the Actor Model is conceived as distributed in space where computational devices called Actors communicate asynchronously using addresses of Actors and the entire computation is not in any well-defined state.

The local state of an Actor is defined when it receives a message and at other times may be indeterminate. With the Actor model you have a configuration-based model of computation. Underpinnings of this model are messages that are received and dynamic by their nature as opposed to fixed.

As you might have already noticed Actor model is scalable by design and by definition. This inherent scalability implies that in near future (as per Reusable Scalable Intelligent Systems by 2025 by Hewitt) we can observe physical implementations of the Actor Mode on computer boxes with millions of Actor cores with average 10-nanosecond latency message passing between quadrillions of Actors.

There also exists a notion of an **Arbiter** in the Actor Model. The Arbiter is what gives us indeterminism. According to the paper Physical Indeterminacy in Digital Computation by Hewitt, arbiters implement actor indeterminacy. In general, the observing fine details by which the order in which an Actor processes messages affects the results. Instead of observing the internal details of arbitration process of Actor computations, we await outcomes. Indeterminacy in arbiters can produce indeterminacy in Actor systems. These arbiters are in fact similar to electronic arbiters . Given an arbiter, you can have multiple inputs (e.g. Input1, Input2) into the arbiter at the same time, but *only* one of the possible outcomes (Output1 or Output2) will come out on the other end. Conceptually Arbiter is some kind of a circuit with logical gates, which can be based on NAND logic (XNOR, NAND and NOT) for instance. These gates require suitable thresholds and other parameters.

After the Arbiter is started, it can remain in a meta-stable state for an unbounded period of time before it finally asserts either Output1 or Output2. But the probability that Arbiter hasn't decided yet decreasing exponentially over time. The internal processes of arbiters are not public processes, they are "black boxes". Attempting to observe them affects their outcomes. Instead of observing arbitration processes' internals, we necessarily await outcomes.

Arbiter is a very convenient and wieldy theoretical framework which was introduced in order to be able to reason or have a description of what is happening when we have more than one message going to the same actor. Abstractly arbiter itself "decides" the ordering of the incoming messages. However a hallmark of the real perceptible world is sundry physical instantiations of a single abstract concept, ergo there might be a lot of instances or examples of Arbiters — a CPU clock generator, Linux kernel timer wheel, CFS, unpredictable pauses caused by garbage collection, OS-level pauses to resolve a major page fault, myriads of network failures, CPU cross calls and TLB shootdowns to name a few.

We may look at the Erlang VM as a concrete exemplar. The semantics of Erlang message passing states that whenever two processes send a message each to a third process, and there is no ordering constraint on the individual send events, we can never rely on which message will end up first in the receiver's mailbox. They could be arbitrarily delayed, even if all processes run within the same Erlang VM. This potentially may happen if senders are on different scheduler threads. The send operations could finish in one order (wall clock time) but the messages may still arrive in swapped order at the receiver's mailbox. Even more can happen if we run a distributed Erlang. For example an Actor sends a message to another Actor in a different subnet over a congested network which is suffering from a TCP incast or a BGP fault. To debunk a potential misconception — message ordering is guaranteed in Erlang only within one process. Namely if there is a live process and we send it a message A and then a message B, it's guaranteed that if message B arrived, message A had arrived before it.

In order to summarise aforesaid we can posit that indeterminacy in Actor Model:

- Is implemented by a digital computation
- Can involve communication with external Actors
- Unbounded nondeterminism: can always halt but take unbounded time.

Actor Model is, in fact, a very abstract conceptual model thus any implementation which is based on the aforementioned axioms of locality is a sound one.

To build an analogy we can assume that actor model is something like a category theory of computation. Category theory is also abstract, minimal and is laser-focused on interactions between objects such as morphisms, functors, natural transformations, transfors without describing the nature and internal structure of the objects themselves.

I'm not good with conclusions hence instead of one I will wrap up with a Robin Milner's quote from his "Elements of Interaction : Turing Award lecture":

Now, the pure lambda-calculus is built with just two kinds of thing: terms and variables. Can we achieve the same economy for a process calculus? Carl Hewitt, with his Actors model, responded to this challenge long ago; he declared that a value, an operator on values, and a process should all be the same kind of thing: an Actor. This goal impressed me, because it implies the homogeneity and completeness of expression ...