Actor Model

A theoretical and practical study

Filippo Fantinato mat. 2041620
filippo.fantinato.2@studenti.unipd.it

July 10, 2023

Contents

List of Figures

Listings

1 Introduction

The Actor Model is a computational model used to design and implement concurrent and distributed systems. It was first proposed by Carl Hewitt in 1973 as a way to model concurrent computation in a more intuitive and modular way than traditional methods. In the Actor Model, an actor is the basic unit of computation which operates independently and communicates with other actors by sending and receiving messages. Actors can be thought of as virtual objects that have their own state, behavior and messaging capabilities. This model has become popular in recent years due to its ability to simplify the development of distributed systems by providing a clear and intuitive abstraction of concurrent computation. Moreover, it allows systems to behave in a way that better matches our mental model.

Actor Model is useful when the problem you are facing involves dependencies and coordinating shared states. The most benefit of using actors and message passing is to avoid explicit locks to protect shared state. It has been implemented in a number of programming languages, including Erlang, Scala, and Akka, and has been used to build a variety of systems, including web servers, distributed databases and video game engines.

In this deepening, I'm introducing the basic concepts of actor model, its advantages and disadvantages, SAL and $A\pi$, two formal languages for actor model, and the actor model in Elixir.

2 What an Actor is

Actors are the basic building block of an Actor Model. Each actor is an autonomous object that operates concurrently and asynchronously, receiving and sending messages to other actors, creating new actors, updating its own local state and designating what to do with the next message. An Actor System consists of a collection of actors, which can send or receive messages from actors outside the system. The messages received by an actor lay in a mailbox and they are processed sequentially by the actor. When an actor isn't performing and has no new messages, it's said idle.
More precisely, when an actor receives a message the following happens:

- the actor adds the message to the end of a queue;
- actor picks the message from the front of the queue;
- actor modifies internal state and sends messages to other actors.

To accomplish this behavior, actors have:

- a mailbox, i.e. the queue where messages end up;
- a behavior, i.e. the state of the actor, internal variables, etc.;
- messages, i.e. pieces of data representing a signal, similar to method calls and their parameters;

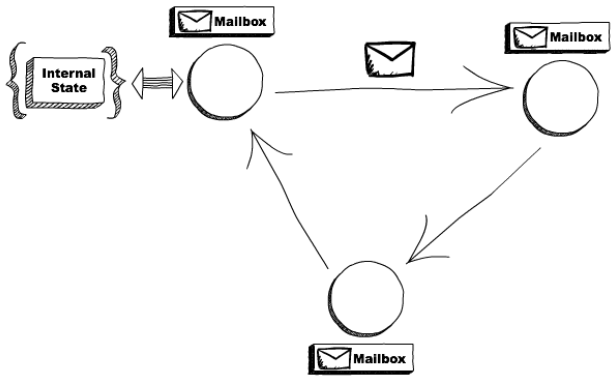

Figure 1: Actor system example

Source: https://www.brianstorti.com/the-actor-model

- an execution environment, i.e. the machinery that takes actors that have messages to react to and invokes their message handling code;
- an address.

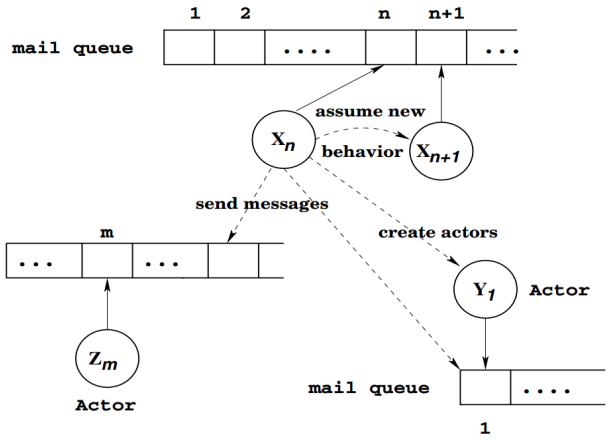

Figure 2: Computation that an actor can perform

Source: [1] An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language

We can group some important properties that actors have:

- uniqueness: each actor has a globally unique ID which is used to communicate with it. Such identifier must be send in a message if someone wants to communicate with that actor;
- persistence: an actor doesn't disappear after processing a message, but it'll wait for new ones;
- encapsulation: such property implies that no actors share state and the state can be modified just by the actor itself. An actor is responsible for exposing safe operations that protect invariant nature of its encapsulated data;

- atomicity: actors handle one message at a time;
- fairness: every actor makes progress if it has some computation to do and every message is eventually delivered to the destination actor;
- location transparency: the actual location of an actor doesn't affect its name, enabling to program without worrying about the actual physical location of them. After termination of an actor, another one may take its name, leading to an important ability called Mobility, which is useful for achieving load-balancing, fault-tolerance and scalable performance.

3 Advantages and Disadvantages

The Actor Model was introduced decades ago as a way to handle parallel processing in a high performance network, an environment that was not available at the time. Today, hardware and infrastructure capabilities have caught up with and Actor Model is recognized as a highly effective solution. Actor Model is used to overcome the limitations of traditional object-oriented programming models and meet the unique challenges of distributed systems.

Problems Actor Model overcomes Traditional OOP languages weren't designed with concurrency and it was very easy to introduce race conditions because of shared state, with the consequent need of locking mechanisms to cope with them. In Figure 3 you can see an example of such problem, since Object2 is shared between Object1 and Object3 and is subject to race conditions. Lock mechanisms are a very costly strategy, since they limit concurrency, requiring heavy-lifting from the operating system to suspend the thread and restore it later, and they block the caller thread, so it cannot do any other work. The latter problem can be solved by running new threads, but these last are also a costly abstraction. Moreover, locks only work well locally and when they come to coordinating across multiple machines the alternative is distributed locks. Distributed lock protocols require several communication round-trips over the network across multiple machines, so latency goes up, and impose a hard limit on scaling out.

Further issues comes when a thread wants to delegate a job to another thread. What usually happens is that the caller puts an object into a memory location shared by the worker thread, allowing the caller to move on and do other tasks. The first issue with that behaviour is how the caller can be notified of the completion of the task, but a more serious issue arises when a task fails with an exception, since we don't really know where the exception propagate to. The caller thread needs to be notified somehow, but there is no an actual call stack to unwind with an exception and one the possible solutions is putting an error code where the caller thread excepts the result once ready. If this notification is not in place, the caller never gets notified of a failure and the task is lost. This situation gets worse when things go really wrong and a worker backed by a thread encounters a bug and ends up in an unrecoverable situation, which makes the thread shut down. The first doubt is who should restart the normal operation of the service hosted by the

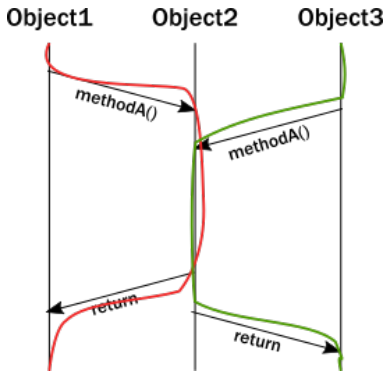

Figure 3: Example of multi-thread execution with Object2 shared
Source: https://doc.akka.io/docs/akka/current/typed/guide/actors-motivation

thread, but especially if there exists a known-good state from which it can be start working. The task is no longer in the shared memory location where tasks are taken from, indeed due to the exception reaching to the top, unwinding all of the call stack, the task state is fully lost. Therefore, we end up having lost a message. That can be seen better in Figure 4, where Main thread didn't take in account the catching of the failure of Worker thread and it will never be notified of the failure.

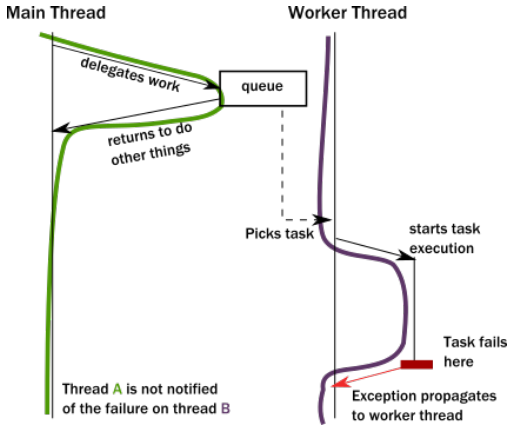

Figure 4: Example of no notification of the worker thread failure
Source: https://doc.akka.io/docs/akka/current/typed/guide/actors-motivation

Finally, another problem is due to the architecture of modern CPUs with many cores. When you write a value on a variable, you're expecting that such value is written in some memory location directly and accessible from any core, but actually CPUs are writing it to cache lines instead of writing to memory directly. Most of these caches are local to the CPU core and it's not visible by others. This feature causes inconsistent states and forces programmers to mark as atomic a variable that is shared by many threads. Anyway, that's a bad practice because atomic forces the writing of the new value on memory and cache lines, which is a very costly operation and represents a bottleneck.

Benefits of Actor Model The Actor Model abstraction allows us to:

- think about your code in terms of communication;
- enforce encapsulation without resorting to locks;
- use the model of cooperative entities reacting to signals, changing state and sending signals to each other to drive the whole application forward.

Instead of calling methods, actors send messages to each other not transferring the thread of execution from the sender to the destination. An actor can send a message and continue without blocking. An important difference of passing messages instead of calling methods is that messages have no return value in general. If it expected a return value, the sending actor would either need to block or to execute the other actor's work on the same thread. Instead, the receiving actor delivers the results in a reply message. Moreover, state of actors is local, not shared and associated to the message passing. This maps to how modern memory hierarchy actually works. In many cases, this means transferring over only the cache lines that contain the data in the message while keeping local state and data cached at the original core.

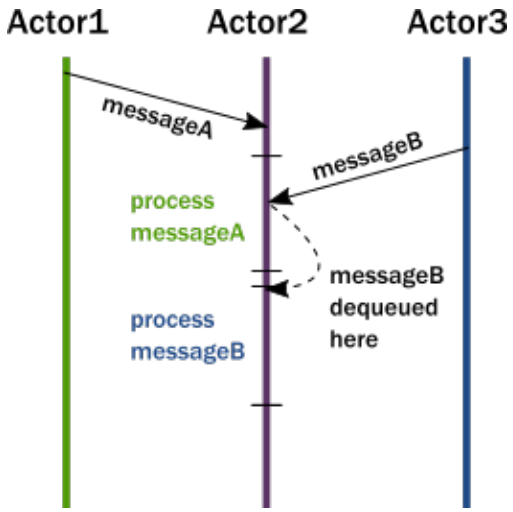

Figure 5: Example of message passing from Actor1 and Actor3 to Actor2
Source: https://doc.akka.io/docs/akka/current/typed/guide/actors-intro

Actors react to messages just like objects to methods invoked on them. The difference is that instead of multiple threads protruding into our actor and wreaking havoc to internal state, actors execute independently from the senders of a message, and they react to incoming messages sequentially, one at a time. While each actor processes messages sent to it sequentially, different actors work concurrently with each other so an actor system can process as many messages simultaneously as many processor cores are available on the machine.

Since we no longer have a shared call stack between actors that send messages to each other, we need to handle error situations differently. There are two kinds of errors we need to consider: the first one is when

the delegate task on the target actor failed due to an error in the task, in which the service actor should reply to the sender with a message showing the error case, while the second one is when a service itself encounters an internal fault. In the latter case, many actor model libraries, such as akka, are organized into a tree-like hierarchy, i.e. an actor that creates another actor becomes the parent of that new actor, so when an actor fails its parent can decide how to react to the failure. Also, if the parent actor is stopped, all of its children are recursively stopped saving their status. The parent becomes the responsible entity for managing a child actor and it's named supervisor. A supervisor strategy is typically defined by the parent actor when it's starting a child actor and it can decide to restart the child actor on certain types of failures or stop it completely on others. Children never go silently dead, instead they are either failing and the supervisor strategy can react to the fault, or they are stopped, in which case interested parties are notified. Restarts are not visible from the outside, so collaborating actors can keep sending messages while the target actor restarts. This feature is very important and invite programmers to split the code logic in more processes in order to separate failures.
Summarizing, Actor model has the following advantages:

- high level of abstraction, since everything in our application is considered as actors;
- vertical and horizontal scalability, given that it's easy to add additional processors or ram to our server and spread a system over many servers;
- fault tolerance and error handling;
- no sharing state and consequently no need of locks.

It has also some for sure disadvantages: mailbox can overflow, susceptible to deadlocks and it's hard to get return values due to the fire and forget, since you have to figure out how to get them with the possibility to add overhead because you have to set up a way to receive it and pass the context through the entire system.

4 SAL: A formal language for Actor Model

SAL (Simple Actor Language) is a basic formal language for the actor model. We can distinguish two main components: configuration and receptionist. A configuration consists of a collection of concurrently executing actors and a collection of messages, while a receptionist is a set of actors in a configuration, more precisely the set of their name, which are visible to the environment. The only way an environment can affect the actors in a configuration is by sending messages to the receptionists of the configuration. Note that the uniqueness of actor names prevents the environment from receiving messages in a configuration that are targeted to the receptionists, since to receive such messages the environment should have an actor with the same name as that of a receptionist. The receptionist set may evolve during interactions, as the messages that the configuration sends to the environment may contain names of actors are not currently in

the receptionist set. SAL has no basic data types or expressions, because to give a formal definition of them they are implemented as $A\pi$-calculus terms, introduced in section 5.

4.1 Syntax

A SAL program consists of a sequence of behaviour definitions followed by a single top level command.

***Def.* 1** (SAL program syntax)**.**

$$prg ::= BDef_1 \dots BDef_n Com$$

4.2 Command definitions

The SAL commands syntax is defined as follows:

***Def.* 2** (Command syntax)**.**

$$\begin{aligned}
Com ::= & send\ [e_1 \dots e_n]\ to\ x && \text{(message send)}\\
& become\ BDef(e_1 \dots e_n) && \text{(new behaviour)}\\
& let\ x\ =\ [recep]\ new\ BDef(e_1 \dots e_k)\ in\ Com && \text{(actor creations)}\\
& if\ e\ then\ Com_1\ else\ Com_2 && \text{(conditional)}\\
& case\ x\ of\ (y_1 : Com_1, \dots, y_n : Com_n) && \text{(name matching)}\\
& Com_1 \mid Com_2 && \text{(composition)}
\end{aligned}$$

In message send term, expressions are evaluated and a message containing the resulting tuple of values is sent to actor x asynchronously. New behaviour specifies a new behavior for the actor which is executing the command. The expressions are evaluated, the results are bound to the parameters in $BDef$ and the resulting closure is the new behaviour of the actor. In Actor creation term, an actor with the specified behaviour and identifier x, which must be unique, is created. There is also the option to mark such actor as receptionist of the program configuration, being able to receive and send messages from and to the environment. The scope of the private actor is only the let scope, while the scope of a receptionist is the entire top level command. Using the conditional term the expression e is evaluated to a boolean. If the result is true, Com_1 is executed, otherwise Com_2 is executed. In name matching term, the name x is matched against the names $y_1 \dots y_n$ and the corresponding command is executed. If there is no matching, then no command is executed. Composition corresponds to the concurrently execution of two sub-commands.

You can notice that there is no notion of sequential composition of commands, since any action is executed concurrently on receiving a message, other than the evaluation order dependencies imposed by the semantics. Moreover, SAL message passing is analogous to call-by-value parameter passing and alternatively a call-by-need message passing scheme can be considered, even though semantically they are equivalent given that there is no recursion.

4.3 Behaviour definitions

The syntax of behaviours definitions is as follows:

***Def.* 3** (Behaviour syntax)**.**

$$BDef ::= def \ < behaviour\ name > \ ([< acquaintance\ list >])[< input\ list >]$$
$$Com$$
$$end\ def$$

The identifier `<behaviour name>` is bound to an abstraction and the scope of this binding is the entire program. The acquaintance list corresponds to the formal parameters of this behaviour, while the input list corresponds to that parameters that a behaviour can receive. The execution of the behaviour body should always result in the execution of at most a single new behaviour command, else such behaviour is said to be erroneous and the corresponding actor is assumed to take a sink behaviour that simply ignores all the messages it receives.

4.4 Factorial example

SAL is not equipped with high-level control flow structures such as recursion and iteration. However, such structures can be encoded as patterns of message passing, like in this example. A request to factorial actor includes a positive integer n and the actor name `cust`, to which the result has to be sent. On receiving a message the actor creates a continuation actor `cont` and sends itself a message with contents $n-1$ and `cont`. The factorial continuation has n and `cust` as its acquaintances. Eventually, a chain of continuation actors will be created each knowing the name of the next in the chain. On receiving a message with an integer, the behaviour of each continuation actor is to multiply the integer with the one it remembers and send the reply to its customer. Note that since the factorial actor is stateless, it can process different factorial requests concurrently without affecting the result of a factorial evaluation.

```
1  def Factorial()[val, cust]
2      become Factorial () |
3      if val == 0
4          then send [1] to cust
5          else let cont = new FactorialCost (val, cust) in
6              send [val-1, cont] to self
7
8  def FactorialCont (val, cust)[arg]
9      send [val*arg] to cust
10 end def
11
12 let x = [recep] new Factorial() in send [5] to x
```

Listing 1: Factorial computation in SAL

5 Aπ-calculus: Actor Model as π-calculus

In this section we represent the Actor model as an asynchronous π-calculus, named $A\pi$. Such interpretation can serve as the basis for actor based concurrent programming languages and it allows to have a formal semantics for SAL by translating its programs into $A\pi$. Moreover, it doesn't provide only a direct basis for comparison between the two models, but also enables us to apply concepts and techniques developed for π-calculus to the actor model.

5.1 Syntax

Assuming an infinite set of names $\mathcal{N}$ ranging over $u, v, w, \ldots$, the set of configurations is defined by the following grammar.

Def. **4** ($A\pi$ syntax)**.**

$$P ::= 0 \mid x(y).P \mid \bar{x}y.P \mid (\nu x)P \mid P_1 \mid P_2 \mid case\ x\ of\ (y_1 : P_1, \ldots, y_n : P_n) \mid B < \tilde{x}; \tilde{y} > P$$

The nil term 0 represents an empty configuration. The output term $\bar{x}y$ represents a configuration with a single message targeted to x and with contents y. The input term $x(y).P$ represents a configuration with an actor x whose behaviour is $(y)P$, where the parameter y is the formal parameter of the behaviour and binds all the free occurrences of y in P. The restricted process $(\nu x)P$ is the same as P, expect for x which is no longer a receptionist of P. Thus, the receptionists of a configuration P are those actors whose names are not bound by a restriction. The composition $P_1 \mid P_2$ is a configuration containing all the actors and messages in P_1, P_2. The configuration $case\ x\ of (y_1 : P_1, \ldots, y_n : P_n)$ behaves like P_i if $x = y_i$ and like 0 otherwise. The term $B < \tilde{x}; \tilde{y} >$ is a behaviour instantiation, where the identifier B has a single defining equation of the form $B = (\tilde{x}; \tilde{y})x_1(z)P$. Such definition provides a template for an actor behaviour. The tuples $\tilde{x}, \tilde{y}$ contain the free names in $x_1(z)P$ and constitute the aquaintance list of the behaviour definition. Such tuples are of length 1 or 2 and x_1 denotes the first component of $\tilde{x}$.

5.2 Operational Semantics

The operational semantics of $A\pi$ is defined using a labeled transition system and it's obtained by simple modifications to the usual rules for asynchronous π-calculus. The transition system is defined modulo alpha-equivalence on processes, i.e. alpha-equivalent processes are declared to the same transitions.

***Def.* 5** ($A\pi$ operational semantics)**.**

$$x(y).P \xrightarrow{xz} P\{z/y\} \quad \text{(inp)}$$

$$\bar{x}y \xrightarrow{\bar{x}y} 0 \quad \text{(out)}$$

$$\frac{P \xrightarrow{\alpha} P' \quad y \notin n(\alpha)}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} \quad \text{(res)}$$

$$\frac{P \xrightarrow{\bar{x}y} P' \quad y \neq x}{(\nu y)P \xrightarrow{\bar{x}y} P'} \quad \text{(open)}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1' \quad bn(\alpha) \cap fn(P_2) = \varnothing}{P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2} \quad \text{(par-1)}$$

$$\frac{P_2 \xrightarrow{\alpha} P_2' \quad bn(\alpha) \cap fn(P_1) = \varnothing}{P_1 \mid P_2 \xrightarrow{\alpha} P_1 \mid P_2'} \quad \text{(par-2)}$$

$$\frac{P_1 \xrightarrow{\bar{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'} \quad \text{(com)}$$

$$\frac{P_1 \xrightarrow{\bar{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2' \quad y \notin fn(P_2)}{P_1 \mid P_2 \xrightarrow{\tau} (\nu y)(P_1' \mid P_2')} \quad \text{(close)}$$

$$case\ x\ of\ (y_1 : P_1, \ldots, y_n : P_n) \xrightarrow{\tau} P_i,\ if\ x = y_i \quad \text{(branch)}$$

$$case\ x\ of\ (y_1 : P_1, \ldots, y_n : P_n) \xrightarrow{\tau} 0,\ if\ x \notin y_1 \ldots y_n \quad \text{(branch-nil)}$$

$$\frac{(x_1(z).P)\{(\tilde{u}, \tilde{v})/(\tilde{x}, \tilde{y})\} \xrightarrow{\alpha} P' \quad B = (\tilde{x}; \tilde{y})x_1(z).P}{B < \tilde{u}; \tilde{v} > \xrightarrow{\alpha} P'} \quad \text{(behaviour)}$$

The interpretation of these terms of the Actor Model is as follows:

- inp: the rule represents the receipt of a message by an actor;
- out: the rule represents the emission of a message;
- res: it states that an action α performed by P can also be performed by $(\nu x)P$, provided that x doesn't occur in α. Such condition disallow the emission of a message which contains the name of a hidden actor in the configuration, that is a non receptionist;
- open: the hidden actor name that is being emitted is bound in the output action, but is no longer bound by a restriction in the transition target, thus the actor which was hidden in the transition source becomes a receptionist in the target;
- par-1, par-2: they capture the concurrent composition of configurations;
- com: it's used to infer the communication of a receptionist name between two composed configurations;
- close: it's used to infer the communication of non-receptionist names between the configurations;

- branch, branch-nil: they capture the selection of a branch in the case statement, whether $x = y_i$ or not;
- behaviour: the rule represents the evolution of a behaviour B given its parameters.

6 Elixir

Elixir is a dynamic, functional programming language built on top of the Erlang Virtual Machine (BEAM). It was created in 2011 by José Valim, a software developer from Brazil, with the goal of providing a modern, scalable and fault-tolerant platform for building distributed applications. Elixir is designed to be concurrent and distributed, which means it can handle large-scale applications with ease. It has a syntax that is similar to Ruby, but it also draws inspiration from other functional programming languages like Erlang and Clojure. One of the unique features of Elixir is its ability to leverage the power of the Erlang Virtual Machine, which has been used for decades to build highly reliable and fault-tolerant systems in industries such as telecommunications and finance. Elixir also provides a wide range of built-in abstractions for handling concurrency, distributed computing, and fault-tolerance, which makes it an ideal choice for building modern web applications and microservices. In summary, Elixir is a modern, functional programming language with a focus on scalability, concurrency, and fault-tolerance. Its powerful abstractions and built-in support for distributed computing make it a great choice for building reliable and high-performance applications.

6.1 Basic syntax

In this section we will do a brief introduction on the Elixir basic syntax. Excepting for the basic data types, such as integers, floats, booleans, atoms and strings, and the basic operations, like arithmetic, Boolean and comparison ones, some notable features are pattern matching, lambda functions, a part of its control structures, Enum and Stream modules.

Pattern Matching Pattern matching in Elixir allows us to match simple values, data structures and even functions. The symbol used for pattern matching is `=` and it's actually called match operator. It matches whether the right and left sides have the same values and, if not, it tries to unify them assigning the values on the right side to the variables on the left side. If the unification fails, a `MatchError` is raised. Some examples can be found here below:

```
iex> x = 1
1
iex> 1 = x
1
iex> 2 = x
```

```
** (MatchError) no match of right hand side value: 1
```

Listing 2: Pattern Matching example 1

```
iex> [head | tail] = [1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

Listing 3: Pattern Matching example 2

```
iex> [head | _] = [1, 2, 3]
iex> head
1
iex> _
* (CompileError) iex:1: invalid use of _. "_" represents a value to be ignored in a pattern
    and cannot be used in expressions
```

Listing 4: Pattern Matching example 3

```
iex> {:ok, result} = {:ok, 42}
{:ok, 42}
iex> {:error, msg} = {:ok, 42}
** (MatchError) no match of right hand side value: {:ok, 42}
```

Listing 5: Pattern Matching example 4

Lambda functions Since Elixir is a functional language, functions are first class citizens. It allows you to write anonymous functions in two ways, the common way, where we specify the arguments and the body separated by ->, enclosed between the keyword fn . . . end, and the &-shorthand way, where just the body is specified after the & symbol and the arguments can be accessed via $\&1, \&2, \&3, \ldots$.

```
iex> sum = fn (a, b) -> a + b end
iex> sum(2, 3)
5

iex> sum = &(&1+&2)
iex> sum(2, 3)
5
```

Listing 6: Lambda functions example 1

Pattern matching isn't limited to just variables, but it can be applied to functions signatures according to the value of their arguments, like in the following examples.

```
iex> handle_result = fn
    {:ok, result} -> IO.puts "Handling result..."
    {:ok, _} -> IO.puts "This would be never run as previous will be matched beforehand."
    {:error} -> IO.puts "An error has occurred!"
end
```

Listing 7: Lambda functions example 2

Case and Cond In addition to the usual control structures `if` and `unless`, Elixir has two extra expressions that can be useful in many cases. Such expressions are: `case`, which is useful to match against multiple patterns and act according to the matching case, and `cond`, needed when instead of matching against values we need to match against conditions like `elif` in many other languages.

```
iex> case {:ok, 42} do
  {:ok, result} when result==42 -> "What is 6*7?" <> result
  {:error} -> "Uh oh!"
  _ -> "Catch all"
end
What is 6*7? 42
```

Listing 8: Case example

```
iex> {x, y} = {2, 3}
iex> cond do
    x+y == 5 -> "x+y is 5"
    x+y == 4 -> "x+y is 4"
    true -> "x+y have wrong values"
end
x+y is 5
```

Listing 9: Cond example

As you can imagine, if there is no match `case` and `cond` will raise an error. To avoid this you can add a default case or condition via `_` symbol or the value `true`.

Enum and Stream modules Enum and Stream modules consist of a set of algorithms for enumerating over enumerable collections, like list and maps. Note that Enum works with eager enumerations, while lazy with lazy ones. Enum and Stream have many functions in common and some notable ones are: maps, all, any, each, filter and reduce. Both Enum and Streams are composable enumerables via the pipe operator.

```
iex> 1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum()
7500000000
```

Listing 10: Enum module example 1

Streams are useful when working with large, possibly infinite, collections. The main difference between Enum and Stream is that Stream allows you to generate streams of data that Enum can work with. Instead of generating intermediate lists, streams build a series of computations that are invoked only when either we pass the underlying stream to the Enum module or we run `Stream.run/1`. As you can see from the examples here above, we invoke a function of Enum module when we are interested in the result of the computation, while `Stream.run/1` when we don't mind the result and the pipeline includes some side-effect operations, like outputting something or writing in a file.

```
iex> odd_numbers = 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(&(rem(&1, 2) != 0))
#Stream<[
  enum: 1..100000,
  funs: [#Function<48.124013645/1 in Stream.map/2>,
   #Function<40.124013645/1 in Stream.filter/2>]
]>
$> Enum.to_list(odd_numbers)
[3, 9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99, 105, 111,
 117, 123, 129, 135, 141, 147, 153, 159, 165, 171, 177, 183, 189, 195, 201, 207,
 213, 219, 225, 231, 237, 243, 249, 255, 261, 267, 273, 279, 285, 291, 297, ...]
```

Listing 11: Stream example 1

```
iex> odd_numbers = 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(&(rem(&1, 2) != 0))
    |> Enum.take(5)
[3, 9, 15, 21, 27]
```

Listing 12: Stream example 2

```
iex> 1..10 |> Stream.map(&(&1 * 3)) |> Stream.filter(&(rem(&1, 2) != 0)) |> Stream.run()
:ok

iex> 1..10 |> Stream.map(&(&1 * 3)) |> Stream.filter(&(rem(&1, 2) != 0)) |> Stream.each(&(IO
    .puts(&1))) |> Stream.run()
3
9
15
21
27
:ok
```

Listing 13: Stream module example 3

6.2 BEAM VM

In order to understand the Actor Model in Elixir, before I need to introduce how the BEAM (Bogdan Erlang Abstract Machine) virtual machine works. That's because what happens at the code level it's strictly related to the VM level.

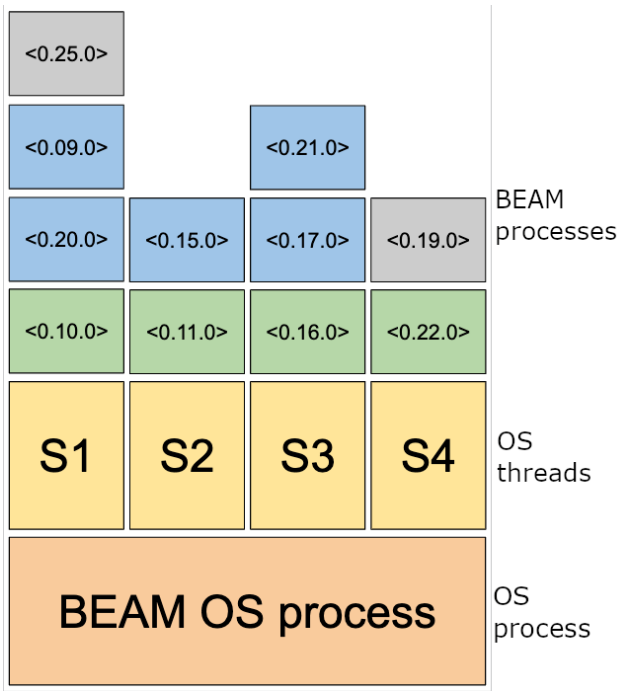

Figure 6: Structure of BEAM VM

Source: https://elixirforum.com/t/how-is-process-in-elixir-different-with-system-thread-in-other-language

The central concept in the Actor Model in Elixir is the notion of process. In the virtual machine, every piece of code runs in a process at runtime. We can say that each process is an independent program, that can communicate with other processes just through messages and executes some code sequentially. As you may have already understood, they correspond to actors in the Actor Model. Processes can be spawned at the code level and all the related functions are explained in subsection 6.3.

Going forward to spawn processes, you will end up having more and more pieces of code which runs independently. What happens at runtime is the key of Elixir speed and lightweight. Each process doesn't actually run in a single OS thread, but the BEAM VM starts as many schedulers as the CPU cores, which are responsible to execute the processes for a certain amount of time. The only OS threads run by a Elixir program are the schedulers and that makes anything lighter and faster. The BEAM processes are internal to the virtual machine, so there is no need of memory areas, data structures and context switching at the operating system level, which are very costly operations, because anything is handled by the virtual machine. From the operating system view, it's running just a program continuously without no need of explicit context switching. The BEAM VM doesn't need to support all the cases an operating system does, so the virtual machine can be specialized just to the required case, allocating less memory and having a more efficient context switcher for processes. That allows us to promote the progress of the system as a whole at the expense of the maximum efficiency of the efficiency of any single activity in the system, indeed the BEAM VM performs a very frequent context preemption, one each 1 millisecond, without having performance issues.

This leads to another reason because splitting the code logic in more actors is advantageous, that is separate latencies and make the system always able to react to any request. From Figure 6 you can see what I just explained. At the bottom there is the BEAM OS process handling some schedules, which are actual OS thread, running some BEAM processes. The green processes are the ones which are being executed, the blue ones are waiting for their turn, while the gray processes are waiting for I/O events, like a database or file read, before going on with their executions.

When we speak about the message passing system, we need to distinguish whether the processes are in the same VM or not. If the actors are in the same VM, the communication is nothing more than copying the messages from an actor to the mailbox of the recipient. If the actors are in different VM, the message is sent though some protocol to the receiver located in another VM. This latter case is analyzed more in detail in the next section.

In order to have always the schedulers working at the same time, there exists a component called load balancer which aims to distribute the work fairly. This component implements the migration logic to allocate processes between the run queues on the schedulers and uses two techniques to balance the load: task stealing and task migration. Task stealing is used every time a scheduler runs out of work and it consists in stealing to other schedulers the processes with higher priority. The task stealing tries to move tasks towards schedulers with lower numbers by trying to steal from schedulers with higher numbers, but since the stealing also will wrap around and steal from schedulers with lower numbers the result is that processes are spread out on all active schedulers. To really exploit the schedulers optimally, the beam virtual machine implements an elaborate migration strategy which, according to statistics gathered during the execution of the system, guides the load balancer to split the processes in a manner to obtain the best performances.

6.3 Actor Model in Elixir

We have already introduced the concept of process in the previous section. Now, let's see all the useful functions to handle processes. The main function to spawn another process is the `spawn/1` function, which takes lambda and returns an identifier of such process. To send a message to another process you can use the `send/2` function, which given the ID of the process and some data, sends such data to it. On the other side, the receiver has a receive block which selects the related behaviour via pattern matching on the message. Processes are actually actors, whose behaviours is defined by the executed lambda. In order to understand how all those functions interact each other, I'm showing you a program to compute the factorial of numbers received in input from a file. First of all, you can observe that I defined two functions: `fact_rec/2` and `fact/1`. That's because the first one is private and it's the actual implementation exploiting tail-recursion, while the latter one is public and avoids wanted behaviours by programmers. Someone could call `fact_rec/2` giving a starting value for the accumulator different from 1.

```
1 defp fact_rec(n, acc\\1)
2 defp fact_rec(n, acc) when n <= 1, do: acc
```

```
defp fact_rec(n, acc) do
    fact_rec(n-1, acc*n)
end

def fact(n) do
    fact_rec(n)
end
```

Listing 14: Factorial example: factorial function

Here below you can find the function `main/1`. It has the aim to open the file given in input from the CLI, spawn the process which reads the file and also call the function `listener/1`. Before going deeper on the last two functions, you can notice an interesting function of the module `Process`: `register/2`. Such function allows you associate a process pid to an atom which will be available globally. Since to send a message to a process its pid is needed, `register/2` avoids the passing of pids weirdly. In this case, I registered the pid of that process with the name `:main`.

```
def main(args \\ []) do
  {:ok, fd} = File.open(args, [:read])
  Process.register(self(), :main)
  spawn(fn () -> read_file(fd) end)
  listener()
end
```

Listing 15: Factorial example: main function

The `read_file/1` takes in put the file descriptor and chooses what to do according the result of the reading. If the message read is `:eof`, it returns just the atom `:ok`. If the message read is an error, then it writes the reason. If anything goes fine, before it converts the data read in integer and then spawns a process which will compute the factorial of the number, sending the result to the process `:main`. Eventually, `read_file/1` is called recursively in order to go through all the file. That's because recursion is preferred to loop constructors. The factorial for each number is computed in different processes, so that anything is concurrent.

```
def read_file(fd) do
    case IO.read(fd, :line) do
        :eof -> :ok
        {:error, reason} -> IO.puts("Error: #{reason}")
        data ->
            case Integer.parse(data) do
            {n, _} ->
                IO.puts("Computing factorial of #{n}")
                spawn(fn () -> send(:main, {:ok, n, fact(n)}) end)
            :error -> IO.puts "#{data} is not a number"
            end
            read_file(fd)
    end
```

```
end
```

Listing 16: Factorial example: function to read a file

The function `listener/0` is aimed to listen all the messaged sent to `:main` and print them. Actually, that's a really basic function and the most interesting piece of code is the call itself at the end. The recursive call is very important and highlights the sequential execution of the code in Elixir. If I hadn't put it, the program would have ended just after the receiving of a message, so it's mandatory in order to read all messages received in the mailbox.

```
def listener() do
    receive do
        {:ok, n, value} ->
            IO.puts("The factorial of #{n} is #{value}")
        _ -> :ok
    end
    listener()
end
```

Listing 17: Factorial example: listener function

Someone may object that the program, as it's written, will never halt. That's true, indeed this problem comes from the fact that all messages are independent. A possible to this problem is to send a `:end` message when `:eof` is reached, but that won't work. Assume to have a big computation, like the factorial of 10000. The `:end` message will be sent for sure before the computation end, so the listener will stop to listen and the message never received.

```
def listener() do
    receive do
    {:ok, n, value} ->
        IO.puts("The factorial of #{n} is #{value}")
        listener()
    :end -> :ok
    _ -> listener()
    end
end

def read_file(fd) do
    case IO.read(fd, :line) do
    :eof -> send(:main, :end)
    ...
    end
end
```

Listing 18: Factorial example: variant with end signal

Actually, there exists two possible solutions to this problem: add a timeout to the receiver or use more advanced functions. The timeout is a horrible and unreliable solution. It can be inserted just by putting a block `after` at the end of the `receive` block, where you can specify what to do after `n` milliseconds. In our case the action to perform is nothing in order make the program end, but that's unreliable since for each timeout we specify, there always exists an computation which could require more time.

```
def listener() do
    receive do
        {:ok, n, value} ->
            IO.puts("The factorial of #{n} is #{value}")
            listener()
        _ -> listener()
    after
        1_000 -> IO.puts("Nothing after 1s")
    end
end
```

Listing 19: Factorial example: variant with timeout to end

The correct solution to that problem is to not use the naive mechanisms for actor model, but exploit some abstraction of them: tasks and agents. Tasks are built on the top of the `spawn/1` function and they are useful when you just need to execute some tasks independently and wait for their results. They are much more powerful and complex, having also an important role in supervision trees. Agents are an abstraction of stateful processes. They provide functions to get and update their state, be instantiated and destroyed. In our case, just tasks are required to overcome the previous problem and a possible solution is shown below. Basically, we create a task for each number and then we wait until their terminations. Note that the functions to wait for the results wait for a limited time and then kill the tasks not ended yet. In order to avoid this, you need to set the timeout to `:infinity`.

```
def main(args \\ []) do
    file = File.stream!(args)
    tasks = file
    |> Stream.map(&String.trim/1)
    |> Stream.map(&String.to_integer/1)
    |> Stream.map(fn (n) -> Task.async(fn () -> {n, FactTasks.fact(n)} end) end)
    |> Enum.to_list()
    res = Task.await_many(tasks, :infinity)
    res |> Enum.each(fn {n, value} -> IO.puts("The factorial of #{n} is #{value}") end)
end
```

Listing 20: Factorial example: variant with tasks

Through the factorial example I can also show what I explained in the BEAM VM section (6.2) just by limiting the number of schedulers. My computer is an octa-core, so by default the BEAM VM runs 8 schedulers and the computation of factorials is spread among them. However, if we cut the number of them

to 1 we can see that the factorial computation is done the same for each number. That's how the BEAM VM works, since the program creates as many processes as factorial to compute and the only scheduler executes alternatively each of them. As it was predictable, the factorial of the bigger number is the last outputted, but you can notice that the computation of smaller numbers doesn't follow any logic and is up to the scheduler.

```
> iex --erl "+S 1:1" -S mix
iex> FactNaive.CLI.main("./numbers.txt")
Computing factorial of 84864
Computing factorial of 45
Computing factorial of 43
Computing factorial of 79
The factorial of 79 is 894618213078297528685144171539831652069808...
The factorial of 45 is 119622220865480194561963161495657715064383733760000000000
The factorial of 43 is 604152630633738356373551320685139975072645120000000000
The factorial of 84864 is ...
```

Listing 21: Output of running the Factorial example limiting the schedulers to 1

Some other functions for concurrency besides the ones above, are: `spawn_link/1` and `spawn_monitor/1`. In particular, `spawn_link/1` is useful when we want to link two processes in order to notify one of them when the other crashes. Normally, when a process crashes also the linked ones crashes, but we can avoid that behaviour just by simply capture the exit message. When we don't want to link two processes, we can use `spawn_monitor/1` which behaves as the previous function but without no risk of crashing for one process if the other crashed. According to whether the linked or monitored processes crash or exit normally, the other process receive a message with `:EXIT` or `:DOWN`. Usually, the catching of the error message `:EXIT` is exploited when there exists a linkage and we don't want to let the other process crash too. Links and monitors play an important role when we are building fault-tolerance systems and they are the naive versions of supervisors, discussed in subsection 6.5.

6.4 Dining Philosophers problem in Elixir

A more complex problem that can show a better usage of actor model in Elixir is the well-known Dining Philosopher problem. The dining philosophers problem states that there are k philosophers sharing a circular table and they eat and think alternatively. There is a plate for each of the philosophers and k forks. A philosopher needs both their right and left forks to eat. A hungry philosopher may only eat if there are both forks available. Otherwise a philosopher puts down their chopstick and begin thinking again. To cope with such problem I used the Lehman and Rabin's solution, which states that any philosopher will eat with probability 1 (i.e. no deadlock and starvation). The solution is: whenever a philosopher is hungry, (s)he tosses a coin to decide whether bringing first the left or right forks. Afterwards, (s)he tries to bring them and, if both forks are free, then starts eating, otherwise (s)he goes on thinking, leaving the possible taken fork.

By modelling the problem in Actor Model, first of all we need to identify the actors. It's easy to see that the actors are two: fork and philosopher. The fork is the easier actor. A fork can be taken, if free, and left by a philosopher, therefore it can receive two types of messages, i.e. request and leave, and send two types of messages, i.e. occupied and taken. At the code level, the Fork module will have the functions `Fork.init/0` and `Fork.loop/1`. The first function is used to spawn the actor, while the second one is what the behaviour of the actor. Clearly, only `Fork.init/0` is public. Note that, following the Elixir philosophy, the actor is stateless and that the information about the state of the Fork is passed as argument at each recursive call.

```
defmodule Fork do
    def init() do
        spawn(fn () -> loop(false) end)
    end

    defp loop(occupied) do
        receive do
            {:request, pid} ->
                if occupied do
                    send(pid, :occupied)
                else
                    send(pid, :taken)
                end
                loop(true)
            :leave ->
                loop(false)
        end
    end
end
```

Listing 22: Dining Philosophers problem: fork actor

The philosopher actor is more complex and it allows me to show advanced concepts. Before going to the main behaviour of the actor, let's see some auxiliary functions. The easier ones are `Philosopher.think/1` and `Philosopher.eat/1`, which encode the behaviour of thinking and eating for a certain amount of time up to 1 second.

```
defp think(name) do
    t = :rand.uniform(1000)
    IO.puts("#{name} is THINKING for #{t} milliseconds")
    :timer.sleep(t)
end

defp eat(name) do
    t = :rand.uniform(1000)
    IO.puts("#{name} is EATING for #{t} milliseconds")
    :timer.sleep(t)
```

```
end
```

Listing 23: Dining Philosophers problem: think and eat functions

Then we have `Philosopher.take_fork/1` and `Philosopher.leave_fork/2`, which send and receive the related messages to a fork actor. The `Philosopher.take_fork/1` function sends the request to pick the given fork, attaching the philosopher pid in order to receive the reply. While, the `Philosopher.leave_fork/1` function sends the message to free the given fork. Note that I don't wait for any message after the sending of the last message, since in my application I assumed that a fork can be free only by the philosopher who owns it at the moment, just to make the application easier.

```
defp take_fork(fork) do
    send(fork, {:request, self()})
    receive do
        :occupied -> :occupied
        :taken -> :taken
        _ -> raise("Something else happens")
    end
end

defp leave_fork(fork) do
    send(fork, :leave)
end
```

Listing 24: Dining Philosophers problem: take and leave functions

Now, let's have a look at the main behaviour of the philosopher actor. At the beginning, a philosopher thinks for a random amount of time. Then, it decides to take either the left or the right fork first. If the first one is occupied, then it waits a random amount of time, it makes the choice of the first fork and tries to take it again. If the first fork is free, then it tries to take the second one. If second one is occupied, then the philosopher leaves the first fork and goes on thinking. While, if also the second fork is free, then it can start eating for a certain amount of time, leaving both forks just finished. At the end, such behaviour is repeated by a recursive call.

```
defp loop(phil) do
    think(phil.name)
    first_right? = :rand.uniform(2) == 1
    {first_fork, second_fork} = case first_right? do
        true  -> {phil.right_fork, phil.left_fork}
        false -> {phil.left_fork, phil.right_fork}
    end
    case take_fork(first_fork) do
        :taken ->
            case take_fork(second_fork) do
                :taken ->
```

```
                    eat(phil.name)
                    leave_fork(second_fork)
                    leave_fork(first_fork)
                :occupied ->
                    leave_fork(first_fork)
            end
        :occupied -> :ok
    end
    loop(phil)
end
```

Listing 25: Dining Philosophers problem: philosopher behaviour

Like for the fork actor, the philosopher is initialized by the only public function `Philosopher.init/3`. In order to make anything tidier, I created a structure philosopher that carries the name, the left fork pid and the right fork pid.

```
defmodule Philosopher do
    defstruct [:name, :left_fork, :right_fork]
    def init(name, left_fork, right_fork) do
        phil = %Philosopher{name: name, left_fork: left_fork, right_fork: right_fork}
        spawn(fn () ->
            loop(phil)
        end)
    end
    ...
end
```

Listing 26: Dining Philosophers problem: philosopher initialization

Finally, we can have a look a the function `DiningPhilosophers.main/1` that starts all the fork and philosopher actors. The execution never ends automatically.

```
def main(_args\\[]) do
    fork1 = Fork.init()
    fork2 = Fork.init()
    fork3 = Fork.init()
    fork4 = Fork.init()
    fork5 = Fork.init()
    phil1 = Philosopher.init("Aristotele", fork1, fork2)
    phil2 = Philosopher.init("Bacon", fork2, fork3)
    phil3 = Philosopher.init("Cartesio", fork3, fork4)
    phil4 = Philosopher.init("Diogene", fork4, fork5)
    phil5 = Philosopher.init("Eliot", fork5, fork1)
    :timer.sleep(:infinity)
end
```

Listing 27: Dining Philosophers problem: main function

6.5 OTP: Open Telecom Platform

OTP is a set of elixir libraries which consists of a number of ready to use components mainly written in Erlang and a set of design principles for programmers. The main components of OTP are: supervisors and distribution.

Supervisors Supervisors are specialized processes with just the purpose to monitor other processes. These supervisors enable us to create fault-tolerant applications by automatically restarting child processes when they fail. They are the core of the Elixir philosophy: let it crash. This doesn't mean that we have to program not being careful about bugs, but we are aware that a live production system has dozens of different reasons why something can go wrong. The disk can fail, memory can be corrupted, bugs, the network may stop working for a second, etc. If we were to write software that attempted to protect or circumvent all of those errors, we would spend more time handling failures than writing our own software. Thus. it's better to let it crash and restart it in a second time.

In order to understand how everything works, let's take a variant of the Dining Philosophers problem seen earlier. What if a philosopher, for the haste to think again, suffocates and dies? That's a perfect problem to show how supervisors works. First of all, let's see the code of the philosopher. What has been changed are the function `Philosopher.eat/1` and `Philosopher.init/3`. In the first function I added the possibility, with probability of 10%, of the philosopher to suffocate. Whenever it suffocates, (s)he before leaves both forks and exits.

```
defp eat(name, fork1, fork2) do
    t = :rand.uniform(1000)
    prob_to_suffocate = :rand.uniform(10)
    if prob_to_suffocate == 1 do
      IO.puts("#{name} is SUFFOCATED!")
      leave_fork(fork1)
      leave_fork(fork2)
      exit(:suffocated)
    end
    IO.puts("#{name} is EATING for #{t} milliseconds")
    :timer.sleep(t)
end
```

Listing 28: Dining Philosopher problem variant: eat function with possibility to suffocate

Regarding the `Philosopher.init/3` function, it has been substituted by `Philosopher.start_link/1` in order to comply with the Elixir standards. In input it receives some options, which contains name, the left fork pid and right fork pid, and returns the atom `:ok` and the pid of the process started no more with `spawn/1`, but with `spawn_link/1`. That's because we need to create a link between the spawned process and the current supervisor one.

```
def start_link(opts) do
```

```
    phil = %Philosopher{name: opts.name, left_fork: opts.left_fork, right_fork: opts.
    right_fork}
    {:ok, spawn_link(fn () -> loop(phil) end)}
end
```

Listing 29: Dining Philosopher problem variant: start_link function of a supervisioned philosofy

Without no supervisor, the program end up to an execution where all philosophers die suffocated. But luckily for them, they have fate on their side that bring each of them back to life whenever it suffocates. So, the supervisor to the philosophers is the fate, which needs some functions to work, i.e. `Fate.start_link/1` and `Fate.init/2`. Both functions accept the list of children, in our case the philosophers and according to some restarting rules, it restarts the crashed ones. The code of the supervisor is as follows.

```
defmodule Fate do
    use Supervisor
    def start_link(philosophers) do
        Supervisor.start_link(__MODULE__, philosophers, name: __MODULE__)
    end
    def init(philosophers) do
        Supervisor.init(philosophers, strategy: :one_for_one, max_restarts: 5)
    end
end
```

Listing 30: Fate supervisior of a philosopher

`Fate.init/2` is the most interesting function, since it defines the restarting rules and instantiate the children. These rules can be:

- `strategy`: one among these restarting strategy: `one_for_one`, i.e. only restart the failed child process, `one_for_all`, i.e. restart all child processes in the event of a failure, and `rest_for_one`, i.e. restart the failed process and any process started after it.
- `max_restarts`: the maximum number of restarts allowed in a time frame after which the supervisor will stop all the children and exit with the signal `:shutdown`;
- `max_seconds`: the time frame in which `max_restarts` applies;
- `name`: a name to register the supervisor process.

On the other hand, `Fate.start_link/1` is less interesting but no less important, since it starts the pipeline to make the supervisor work.

In order to be started, a supervisor child must specify some information about itself. Those information are located in the `child_spec/1` function and here below you can find the function defined for the philosopher. In input it accepts the options and in output it return a map with the following fields:

- `id`: used by the supervisor to identify the child specification;

- `start`: the Module/Function/Arguments to call when started by the supervisor;
- `restart`: how to handle the restarting of a child. It can be a value among: `:permanent`, i.e. child is always restarted, `:temporary`, i.e. child process is never restarted, and `:transient`, i.e. child process is restarted only if it terminates abnormally;
- `shutdown`: defines child's behavior during shutdown. It can be a value among: `:brutal_kill`, i.e. stops immediately, `n`, i.e. time in milliseconds supervisor will wait before killing child process, and `:infinity`, i.e. supervisor will wait indefinitely before killing child process;
- `type`: it can be either `:worker` or `:supervisor`.

```
def child_spec(opts) do
    %{
      id: opts.name,
      start: {__MODULE__, :start_link, [opts]},
      restart: :permanent,
      shutdown: 5_000,
      type: :worker
    }
end
```

Listing 31: Dining Philosopher problem variant: child_spec function

Finally, we can see the main function where the supervisor starts.

```
def main(_args\\[]) do
    fork1 = Fork.init()
    fork2 = Fork.init()
    fork3 = Fork.init()
    fork4 = Fork.init()
    fork5 = Fork.init()
    philosophers = [
      {Philosopher, %{name: "Aristotele", left_fork: fork1, right_fork: fork2}},
      {Philosopher, %{name: "Bacon", left_fork: fork2, right_fork: fork3}},
      {Philosopher, %{name: "Cartesio", left_fork: fork3, right_fork: fork4}},
      {Philosopher, %{name: "Diogene", left_fork: fork4, right_fork: fork5}},
      {Philosopher, %{name: "Ernesto", left_fork: fork5, right_fork: fork1}}
    ]
    Fate.start_link(philosophers)
    :timer.sleep(:timer.minutes(3))
end
```

Listing 32: Dining Philosopher problem variant: main function

Clearly, that's a stupid example where the children are forced to crash and there is just one supervisor. You can create more complex applications with many supervisors which can also be supervised by other supervisors, end up having a supervision tree. Supervision tree are the key to build fault-tolerance application,

as failure is isolated to some branch of the tree, where the supervisor of that branch can either restart the failed children, or allow itself to fail as well, bubbling the failure up to the next highest supervisor in the tree. According to whether the processes to supervise are static or dynamic, we can distinguish two types of supervisors. The static supervisor is the easier case and it requires the list of children and the strategy.

Distribution We can run our Elixir applications on a set of different nodes distributed across a single host or across multiple hosts. Elixir allows us to communicate across these nodes via a few different mechanisms. This topic is too complex to be addressed, so I'm introducing you only the main concepts.

In order to communicate with some Elixir node, we need to run the applications with some unique name which identifies the node in the network. In order to do that, we can run the VM by passing the argument `--sname [NAME]`. Then, if we know a node name, we can communicate with that and execute some functions on that machine just by calling `Node.spawn/2`, `Node.spawn_link/2` and `Node.spawn_monitor/2`. These last functions receive in input the name of the node and the code to execute in that node. As example, let's start our application computing the factorial in a distributed node named alice and let's connect to alice from another node named bob. Before, the node bob needs to spawn a process on the node alice to which sends requests, because the factorial application doesn't provide it. So then bob can send a message to that node and receive the response.

```
$ iex --sname bob
iex(bob@localhost)> pid = Node.spawn_link(:"alice@localhost", fn () ->
                          receive do
                              {:fact, n, pid} -> send(pid, FactNaive.fact(n))
                              _ -> :ok
                          end
                      end)
iex(bob@localhost)> send(pid, {:fact, 5, self()})
iex(bob@localhost)> receive do n -> IO.puts(n) end
120
:ok
```

Listing 33: Example of distributed communication in Elixir

As happened for the basic functions for concurrency, there are three advanced alternatives based to `Node.spawn_link/2` that we could use in our implementation:

- we could use Erlang `:erpc` module to execute functions on a remote node;
- we could have a server running on the other node and send requests to that node via the GenServer API;
- we could use tasks that can be spawned on both local and remote nodes.

Something that you can notice is that Elixir ships with facilities to connect nodes and exchange information between them. In fact, we use the same concepts of processes, message passing and receiving messages when working in a distributed environment because Elixir processes are location transparent.

7 Conclusions

Concluding, the Actor Model is a powerful paradigm for building scalable, fault-tolerant and concurrent systems. It provides a high level of abstraction that allows for modular design and efficient use of resources. Some notable features are:

- scalability: the Actor Model is highly scalable, as it allows for the creation of many actors that can run concurrently. Each actor is independent of the others, and communication between actors is done through message passing, which can be done asynchronously;
- fault-tolerance: the Actor Model provides a high degree of fault-tolerance, as each actor is responsible for its own state and processing. If an actor fails, it can be restarted without affecting the other actors in the system;
- modularity: the Actor Model provides a modular approach to system design. Each actor can be developed independently, and can be reused in different contexts;
- concurrency: the Actor Model provides a natural way to express concurrency, as each actor can run independently of the others. This allows for efficient use of resources and can lead to improved performance;
- message-passing: message passing is the primary means of communication in the Actor Model. This provides a clear and simple way to express communication between actors.

References

[1] Gul Agha and Prasanna Thati. *An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language*. Ed. by Olaf Owe, Stein Krogdahl, and Tom Lyche. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 26–57. ISBN: 978-3-540-39993-3. DOI: 10.1007/978-3-540-39993-3_4. URL: https://doi.org/10.1007/978-3-540-39993-3_4.

[2] *Elixir official documentation*. URL: https://elixir-lang.org/docs.html.

[3] *Elixir school*. URL: https://elixirschool.com/en.

[4] *The Soul of Erlang and Elixir*. URL: https://www.youtube.com/watch?v=JvBT4XBdoUE.

[5] *Why modern systems need a new programming model*. URL: https://doc.akka.io/docs/akka/current/typed/guide/actors-motivation.html#the-challenge-of-encapsulation.

[6] *How the Actor Model Meets the Needs of Modern, Distributed Systems*. URL: https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html.

[7] *Elixir and The Beam: How Concurrency Really Works*. URL: https://medium.com/flatiron-labs/elixir-and-the-beam-how-concurrency-really-works-3cc151cddd61.

[8] *How to build a self-healing system using supervision tree in Elixir*. URL: https://kodius.com/blog/elixir-supervision-tree.

[9] *Using Supervisors to Organize Your Elixir Application*. URL: https://blog.appsignal.com/2021/08/23/using-supervisors-to-organize-your-elixir-application.html.