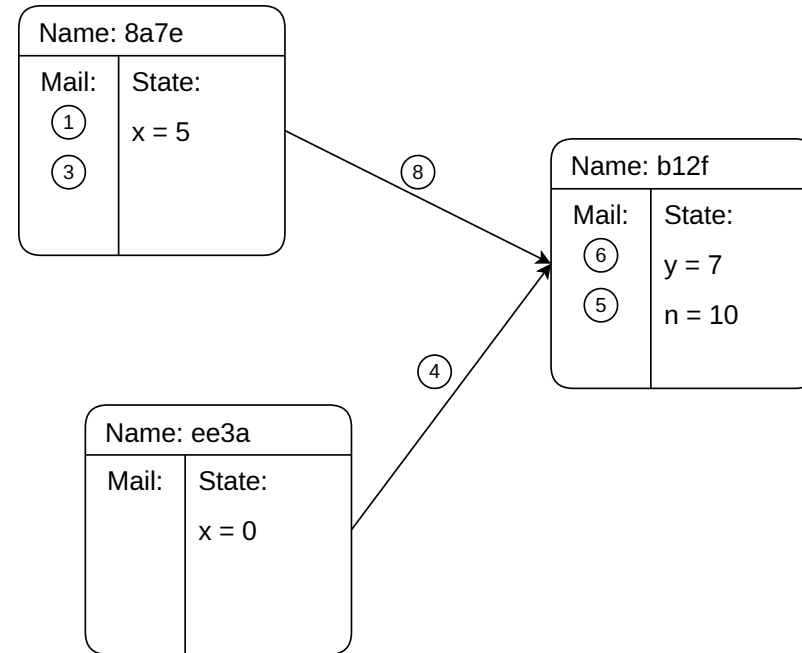


# Actors

# Actors Today

An actor has:

- a unique name
- a local state
- an unbounded mailbox
- a description of its behavior



# The ACTOR Formalism

This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams].

[...]

The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism.

[...]

The architecture is general with respect to control structure and does not have or need goto, interrupt, or semaphore primitives.

Intuitively, an ACTOR is an active agent which plays a role on cue according to a script.

[...]

Data structures, functions, semaphores, monitors, ports, descriptions, Quillian nets, logical formulae, numbers, identifiers, demons, processes, contexts, and data bases can all be shown to be special cases of actors.

# Patterns of Passing Messages

In this paper we present an approach to modelling intelligence in terms of a society of communicating knowledge-based problem-solving experts.

At a more superficial and imprecise level, each actor may be thought of as having two aspects which together realize the behavior which it manifests:

the ACTION it should take when it is sent a message;

its ACQUAINTANCES which is the finite collection of actors that it directly KNOWS ABOUT.

Carl Hewitt. 1977. Viewing control structures as patterns of passing messages.

# Actors: a model of concurrent computation

An actor may be described by specifying:

- its mail address, to which there corresponds a sufficiently large mail queue; and,
- its behavior, which is a function of the communication accepted.

Actors are computational agents which map each incoming communication to a 3-tuple consisting of:

1. a finite set of communications sent to other actors;
2. a new behavior (which will govern the response to the next communication processed); and,
3. a finite set of new actors created

Gul Agha. 1986. Actors: a model of concurrent computation in distributed systems.

## When to use them

- Distributed systems
- Servers and services
- Telecom and signalling systems
- Chat and messaging systems
- Event-driven microservices
- Online gaming backends

# Elixir

<https://elixir-lang.org>

```
"Elixir" |> String.graphemes() |> Enum.frequencies()
```

Designed by José Valim, first appeared 2012

<https://www.erlang.org>

```
fact(0) -> 1;  
fact(N) -> N * fact(N-1).
```

```
example:fact(10).
```

Based on Erlang, designed by Joe Armstrong, started 1986, open-sourced 1998

**demo/basics.exs**



**demo/filesystem.exs**

# Actors in Erlang/Elixir

Two processes operating on the same machine must be as independent as if they ran on physically separated machines.

[...]

1. Processes have "share nothing" semantics. This is obvious since they are imagined to run on physically separated machines.
2. Message passing is the only way to pass data between processes. [...]
3. Isolation implies that message passing is asynchronous. [...]
4. Since nothing is shared, everything necessary to perform a distributed computation must be copied. [...]

Joe Armstrong. 2003. Making reliable distributed systems in the presence of software errors.

**demo/processes.exs**

# Names of Actors

We require that the names of processes are unforgeable. This means that it should be impossible to guess the name of a process, and thereby interact with that process. We will assume that processes know their own names, and that processes which create other processes know the names of the processes which they have created.

[...]

In many primitive religions it was believed that humans had powers over spirits if they could command them by their real names. Knowing the real name of a spirit gave you power over the spirit, and using this name you could command the spirit to do various things for you

Joe Armstrong. 2003. Making reliable distributed systems in the presence of software errors.

# PIDs in Elixir

<https://hexdocs.pm/elixir/Kernel.html#self/0>

```
self()
```

Returns the PID (process identifier) of the calling process.

<https://hexdocs.pm/elixir/Kernel.html#spawn/1>

```
spawn(fun)
```

Spawns the given function and returns its PID.

**demo/names.exs**

# Message Passing

Message passing obeys the following rules:

1. Message passing is assumed to be atomic which means that a message is either delivered in its entirety or not at all.
2. Message passing between a pair of processes is assumed to be ordered meaning that if a sequence of messages is sent and received between any pair of processes then the messages will be received in the same order they were sent.
3. Messages should not contain pointers to data structures contained within processes — they should only contain constants and/or Pids.

We say that such message passing has *send and pray semantics*. We *send* the message and *pray* that it arrives.

# Message Passing in Elixir

<https://hexdocs.pm/elixir/Kernel.html#send/2>

```
send(dest, message)
```

Sends a message to the given dest and returns the message.

<https://hexdocs.pm/elixir/Kernel.SpecialForms.html#receive/1>

```
receive(args)
```

Checks if there is a message matching any of the given clauses in the current process mailbox.



**demo/messages.exs**

**demo/order.exs**

# Message Buffers are Unbounded

In a purely physical context, the finiteness of the universe suggests that a communication sent ought to be delivered.

[...]

There are, realistically, no unbounded buffers in the physically realizable universe. This is similar to the fact that there are no unbounded stacks in the universe, and certainly not in our processors, and yet we parse recursive control structures in algolic languages as though there were an infinite stack.

The alternate to assuming unbounded space is that we have to assume some specific finite limit; but each finite limit leads to a different behavior. There is, however, no general limit on buffers: the size of any real buffer will be specific to any particular implementation and its limitations.

Gul Agha. 1986. Actors: a model of concurrent computation in distributed systems.

**demo/overflow.exs**

# Changing State

We tail-call the actor with another argument.

```
def loop(count) do
  receive do
    :increment ->
      loop(count + 1)
  end
end
```

**demo/increment.exs**

**demo/state.exs**

# Reply-to Addresses

Often the envelope of a messenger is a REQUEST which In addition to a request message contains an actor *c* to which a reply to the request should be sent. Such an envelope is packaged as follows:

(request: the-message (reply-to: *c*))

The ACTOR *c* Is closely related to the continuation FUNCTIONS used by Morris, Wadsworth, Reynolds, and Strachey.

An ordinary functional call to a function *f* with arguments *arg*<sub>1</sub>, ..., through *arg*<sub>*k*</sub> is implemented in PLASMA by passing to *f* a request envelope with a message consisting of the tuple [*arg*<sub>1</sub>, ..., *arg*<sub>*k*</sub>] of arguments and a continuation actor to which the value of *f* should be sent.

Carl Hewitt. 1977. Viewing control structures as patterns of passing messages.



**demo/reply.exs**

# Correlation Tokens

[https://hexdocs.pm/elixir/Kernel.html#make\\_ref/0](https://hexdocs.pm/elixir/Kernel.html#make_ref/0)

```
make_ref()
```

Returns an almost unique reference.

**demo/correlation.exs**

**demo/chat\_client.exe**

**demo/chat\_server.exs**