OTP

Anton Fagerberg

anton.fagerberg@jayway.com

What is OTP?

> OTP as a complete development environment for concurrent programming.

Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

OTP is

- The Erlang interpreter and compiler
- Erlang standard libraries
- Dialyzer, a static analysis tool
- Mnesia, a distributed database
- Erlang Term Storage (ETS), an in-memory database
- A debugger
- An event tracer
- A release-management tool

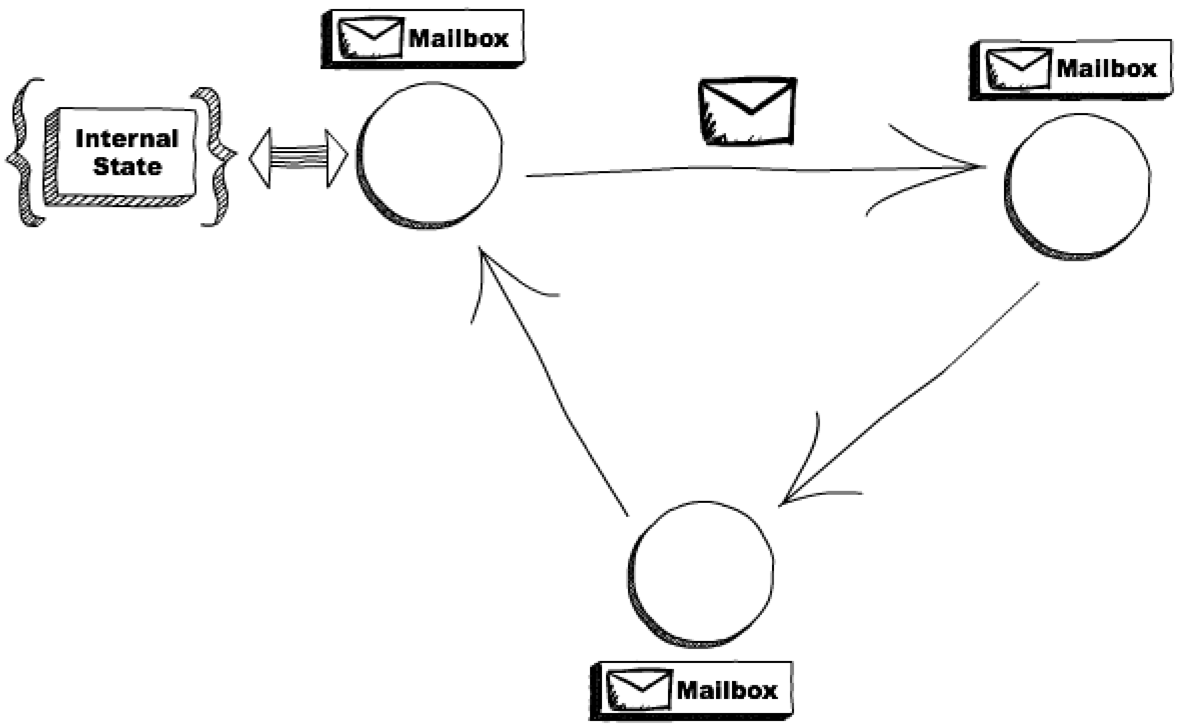
Benjamin Tan Wei Hao. "The Little Elixir & OTP Guidebook."

OTP Behaviors

- Design patterns for actors
- Provide generic pieces

Actor

- Concurrency primitive
- Each actor is a process
- Message passing (only interaction)
- No shared information (memory) with other actors



http://www.brianstorti.com/the-actor-model/

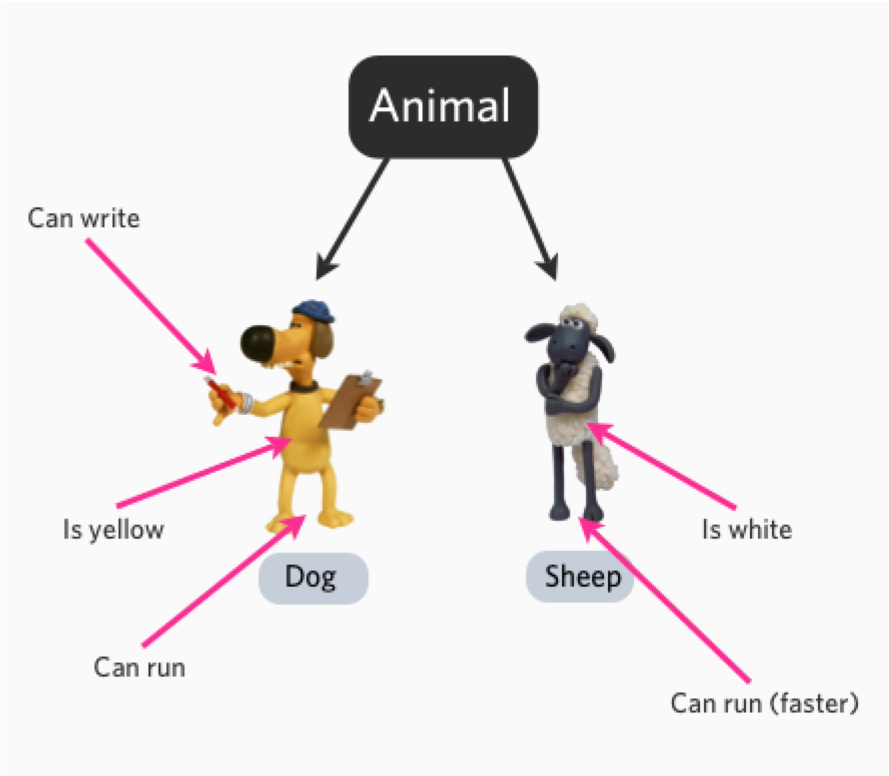
What actors do

When an actor receives a message, it can do one of these 3 things:

1. Create more actors.
2. Send messages to other actors.
3. Designates what to do with the next message.

http://www.brianstorti.com/the-actor-model/

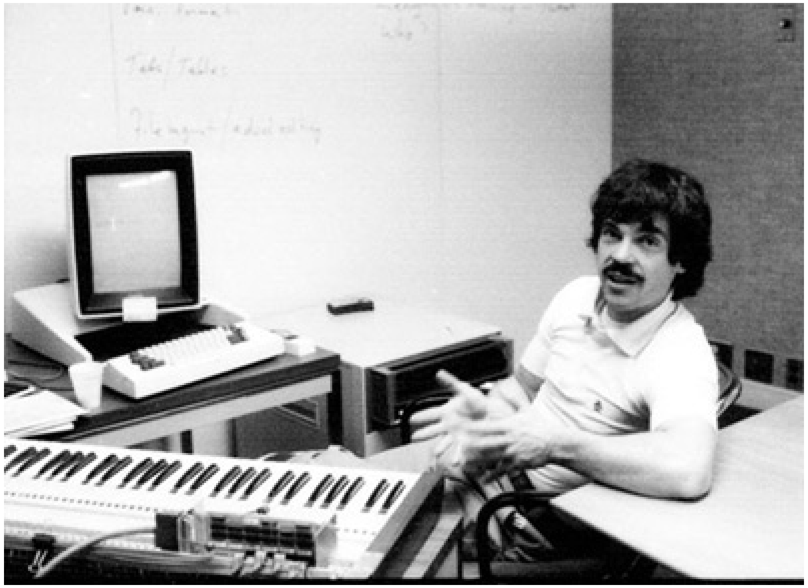
Detour into Object Orientation



http://www.codercaste.com/2011/01/12/what-is-object-oriented-programming-and-why-you-need-to-use-it/

Alan Kay

Coined the term "object orientation"



> He is best known for his pioneering work on object-oriented programming and windowing graphical user interface design.

The Early History of Smalltalk

March 1993

http://stephane.ducasse.free.fr/FreeBooks/SmalltalkHistoryHOPL.pdf

The Early History of Smalltalk

printed on t-shirts), and severely restricted the kinds of publications that could be made. This was particularly disastrous for LRC, since we were the "lunatic fringe" (so-called by the other computer scientists), were planning to go out to the schools, and needed to share our ideas (and programs) with our colleagues such as Seymour Papert and Don Norman.

Executive "X" apparently heard some harsh words at Stamford about us, because when he returned around Christmas and found out about the interim Dynabook, he got even more angry and tried to kill it. Butler wound up writing a masterful defence of the machine to hold him off, and he went back to his "task force".

Chuck had started his "bet" on November 22, 1972. He and two technicians did all of the machine except for the disk interface which was done by Ed McCreight. It had a ~500,000 pixel (606x808) bitmap display, its microcode instruction rate was about 6MIPs, it had a grand total of 128k, and the entire machine (exclusive of the memory) was rendered in 160 MSI chips distributed on two cards. It was beautiful [Thacker,1972, 1986]. One of the wonderful features of the machine was "zero-over-head" tasking. It had 16 program counters, one for each task. Condition flags were tied to interesting events (such as "horizontal retrace pulse", and "disk sector pulse", etc.). Lookaside logic scanned the flags while the current instruction was executing and picked the highest priority program counter to fetch from next. The machine never had to wait, and the result was that most hardware functions (particularly those that involved i/o (like feeding the display and handling the disk) could be replaced by microcode. Even the refresh of the MOS dynamic RAM was done by a task. In other words, this was a coroutine architecture. Chuck claimed that he got the idea from a lecture I had given on coroutines a few months before, but I remembered that Wes Clark's TX-2 (the Sketchpad machine) had used the idea first, and I probably mentioned that in the talk.

BILBO, the first "Interim Dynabook", and Cookie Monster", the first graphics it displayed. April, 1973

In early April, just a little over three months from the start, the first Interim Dynabook, known as 'Bilbo,' greeted the world and we had the first bit-map picture on the screen within minutes: the Muppets' Cookie Monster that I had sketched on our painting system.

Soon Dan had bootstrapped Smalltalk across, and for many months it was the sole software system to run on the Interim Dynabook. **Appendix I** has an "acknowledgements" document I wrote from this time that is interesting in its allocation of credits and the various priorities associated with them. My $230K was enough to get 15 of the original projected 30 machines (over the years some 2000 Interim Dynabooks were actually built). True to Schopenhauer's observation, Executive "X" now decided that the Interim Dynabook was a good idea and he wanted all but two for his lab (I was in the other lab). I had to go to considerable lengths to get our machines back, but finally succeeded.

1. Everything is an *object*
2. Objects communicate by sending and receiving *messages* (in terms of objects)
3. Objects have their *own memory* (in terms of objects)
4. Every object is an *instance* of a *class* (which must be an object)
5. The class holds the shared *behavior* for its instances (in the form of objects in a program list)
6. To eval a program list, control is passed to the first object and the remainder is treated as its message

By this time most of Smalltalk's schemes had been sorted out into six main ideas that were in accord with the initial premises in designing the interpreter. The first three principles are what objects "are about"—how they are seen and used from "the outside". These did not require any modification over the years. The last three—objects from the inside—were tinkered with in every version of Smalltalk (and in subsequent OOP designs). In this scheme (1 & 4) imply that classes are objects and that they must be instances of themself. (6) implies a LISPlike universal syntax, but with the receiving object as the first item followed by the message. Thus c_i <- de (with subscripting rendered as "∘" and multiplication as "*") means:

receiver	message
c	∘ *i* <- *d*e*

The *c* is bound to the receiving object, and all of∘ *i* <- *d*e* is the message to it. The message is made up of a literal token "∘", an expression to be evaluated in the sender's context (in this case *i*), another literal token <-, followed by an expression to be evaluated in the sender's context (*d*e*). Since "LISP" pairs are made from 2 element objects they can be indexed more simply: *c hd*, *c tl*, and *c hd <- foo*, etc.

"Simple" expressions like *a+b* and 3+4 seemed more troublesome at first. Did it really make sense to think of them as:

receiver	message
a	+ *b*
3	+ 4

It seemed silly if only integers were considered, but there are many other metaphoric readings of "+", such as:

"kitty" | + "kat" => "kittykat"

$$\begin{bmatrix}3&4&5\\6&7&8\end{bmatrix} \quad | + 4 \quad => \begin{bmatrix}7&8&9\\10&11&12\end{bmatrix}$$

This led to a style of finding *generic behaviors* for message symbols. "Polymorphism" is the official term (I believe derived from Strachey), but it is not really apt as its original meaning applied only to functions that could take more than one type of argument. An example class of objects in Smalltalk-72, such as a model of CONS pairs, would look like:

to likeLOGO, except makes a class from its message

ISNEW is true if a new instance has been created

true any object not false acts as true

true ⇒ m n will evaluate m and escape from surrounding ( )

false ⇒ m n will evaluate n

temporary variable

Instance variables

```
to Pair b | h t                                    "b is temp. h, t are internal instance vars"
   (ISNEW   ⇒ (:h. :t)                              "cons—if no explicit return is given, SELF is returned
    ◻hd     ⇒ (◻<- ⇒ (^:h)^h)                       "replaca and car"
    ◻tl     ⇒ (◻<- ⇒ (:t)^t)                        "replacd and cdr"
    ◻isPair ⇒ (^true)
    ◻print  ⇒ ('(print. SELF mprint)
    ◻mprint ⇒ (h print. t isNil ⇒ (') print) t isPair ⇒ (t mprint) ⇒ '• print. t print. ') print)
    ◻length ⇒ (t isPair ⇒ (^1+t length) 1))
```

: evals the next part of message and binds result to the variable in its message

◻ eyeball looks to see if its message is a literal token in the message stream

^ send-back returns its value to sender

. "statement separator" value is following message

Since control is passed to the class before any of the rest of the message is considered—the class can decide not to receive at its discretion—complete protection is retained. Smalltalk-72 objects are "shiny" and impervious to attack. Part of the environment is the binding of the SENDER in the "messenger object" (a generalized activation record) which allows the receiver to determine differential privileges (see **Appendix II** for more details). This looked ahead to the eventual use of Smalltalk as a network OS (see [Goldstein & Bobrow 1980]), and I don't recall it being used very much in Smalltalk-72.

One of the styles retained from Smalltalk-71 was the comingling of function and class ideas. In other works, Smalltalk-72 classes looked like and could be used as functions, but it was easy to produce an instance (a kind of closure) by using the object ISNEW. Thus factorial could be written "extensionally" as:

*to fact n (^if :n=0 then 1 else n*fact n-1)*

or "intensionally", as part of class integer:

(... ◻! ⇒ (^:n=l) ⇒ (1) (n-1)!)

Of course, the whole idea of Smalltalk (and OOP in general) is to define everything *intensionally*. And this was the direction of movement as we learned how to program in the new style. I never liked this syntax (too many parentheses and nestings) and wanted something flatter and more gram-

The first three principles are what objects "are about"

1. Everything is an *object*
2. Objects communicate by sending and receiving *messages* (in terms of objects)
3. Objects have their *own memory* (in terms of objects)

1. Everything is an *object*
2. **Objects communicate by sending and receiving *messages***
3. **Objects have their *own memory***

Back to OTP

OTP Behaviors

- GenServer
  - Implementing the server of a client-server relationship
- Supervisor
  - Implementing supervision functionality
- Application
  - Working with applications and defining application callbacks

Elixir is creating more - and you can implement your own!

GenServer (Generic Server)

Abstraction of client / server functionality.

GenServer

Provides

- Start (spawn) server process
- Maintain state in server
- Handle requests, send responses
- Stopping server process
- Naming conventions
- Handle unexpected messages
- Consistent structure

GenServer

Leaves you to define

- State to initialize
- What messages to handle (requests)
- When to reply (async / sync)
- What messages to reply with
- What resources to clean-up on termination

```elixir
def loop(results \\ [], results_expected) do
  receive do
    {:ok, result} ->
      new_results = [result|results]
      loop(new_results, results_expected)

    _ ->
      loop(results, results_expected)
  end
end
```

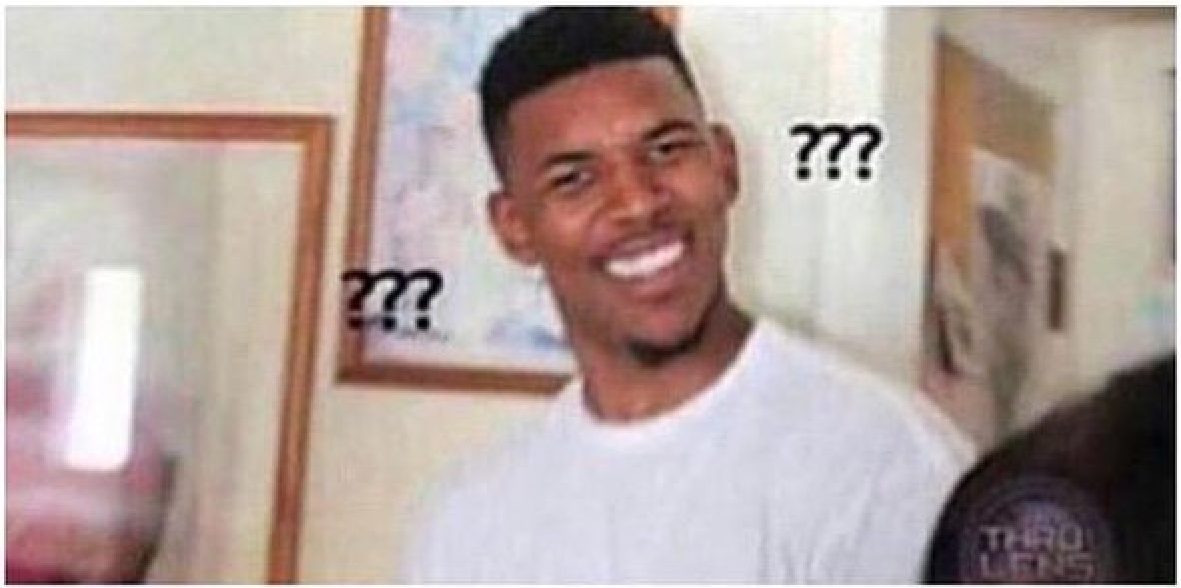
Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

```elixir
def handle_call({:location, location}, _from, state) do
  new_state = update_stats(stats, location)
  {:reply, "hello!", new_state}
end
```

Sequential programs

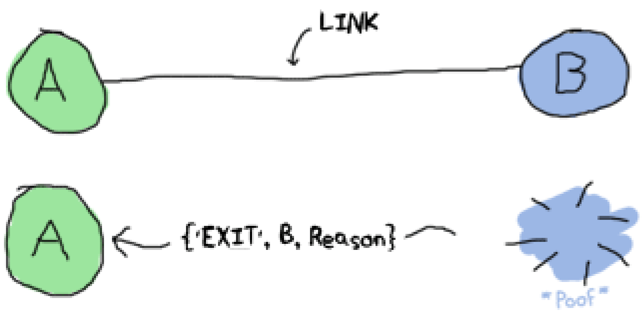
- Typically one main process
- Program defensively
- `try` & `catch`
- `if err != nil`

Let it crash!



Link

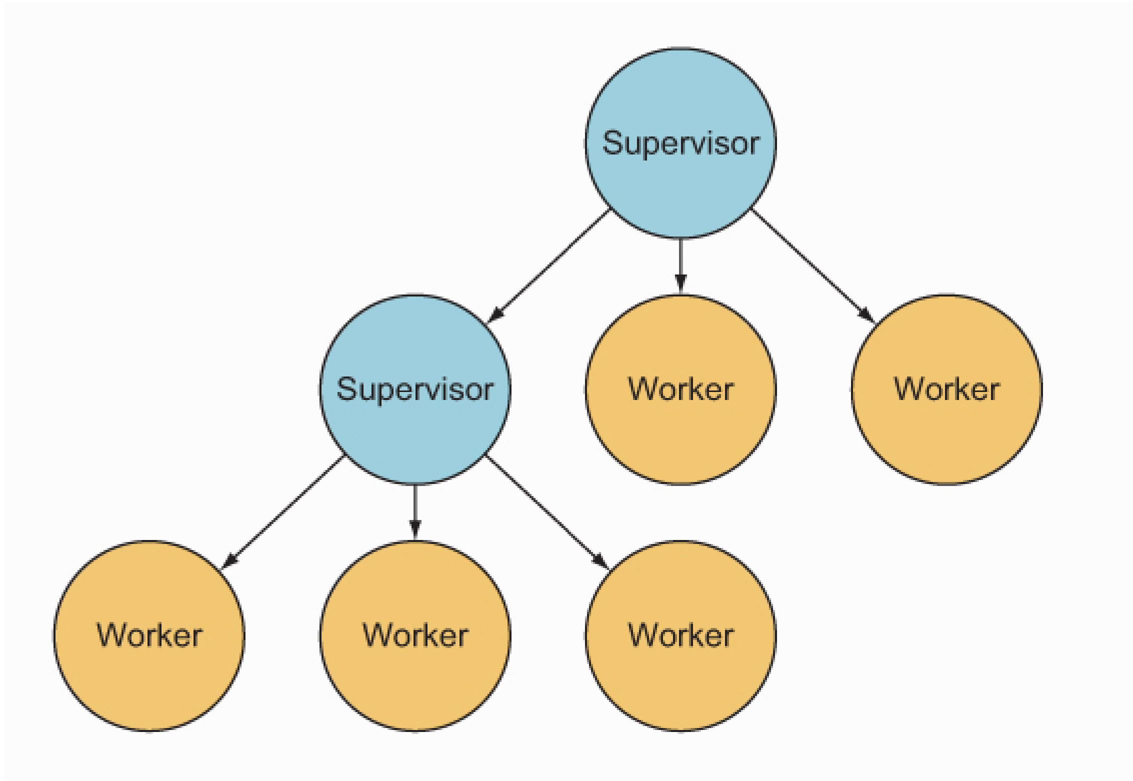
- Actors can link themselves to other actors
- (or monitor them)



http://learnyousomeerlang.com/errors-and-processes

Supervisors

- Observe other processes
- Take action when things break
- GenServer makes it easy to be supervised



Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

Let it crash

- Delegate error detection and handling to other actors
- Do not code defensively

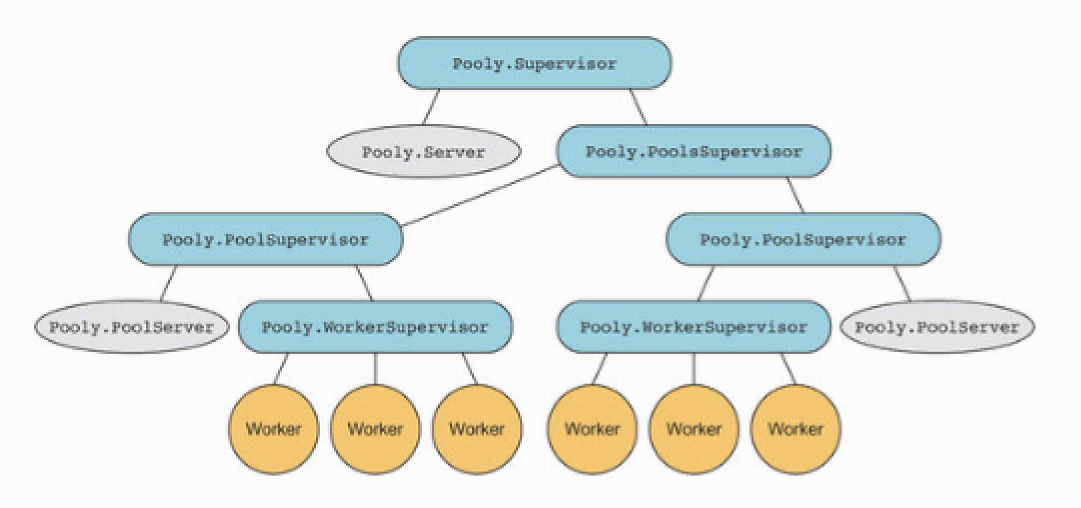
Restart stratergies

- one for one
  - If a process dies, only that process is restarted.
- one for all
  - All process in the supervision tree dies with it.
- rest for one
  - Processes started *after* the failing process are terminated.
- simple one for one
  - Factor method, many instances of same process.

Also

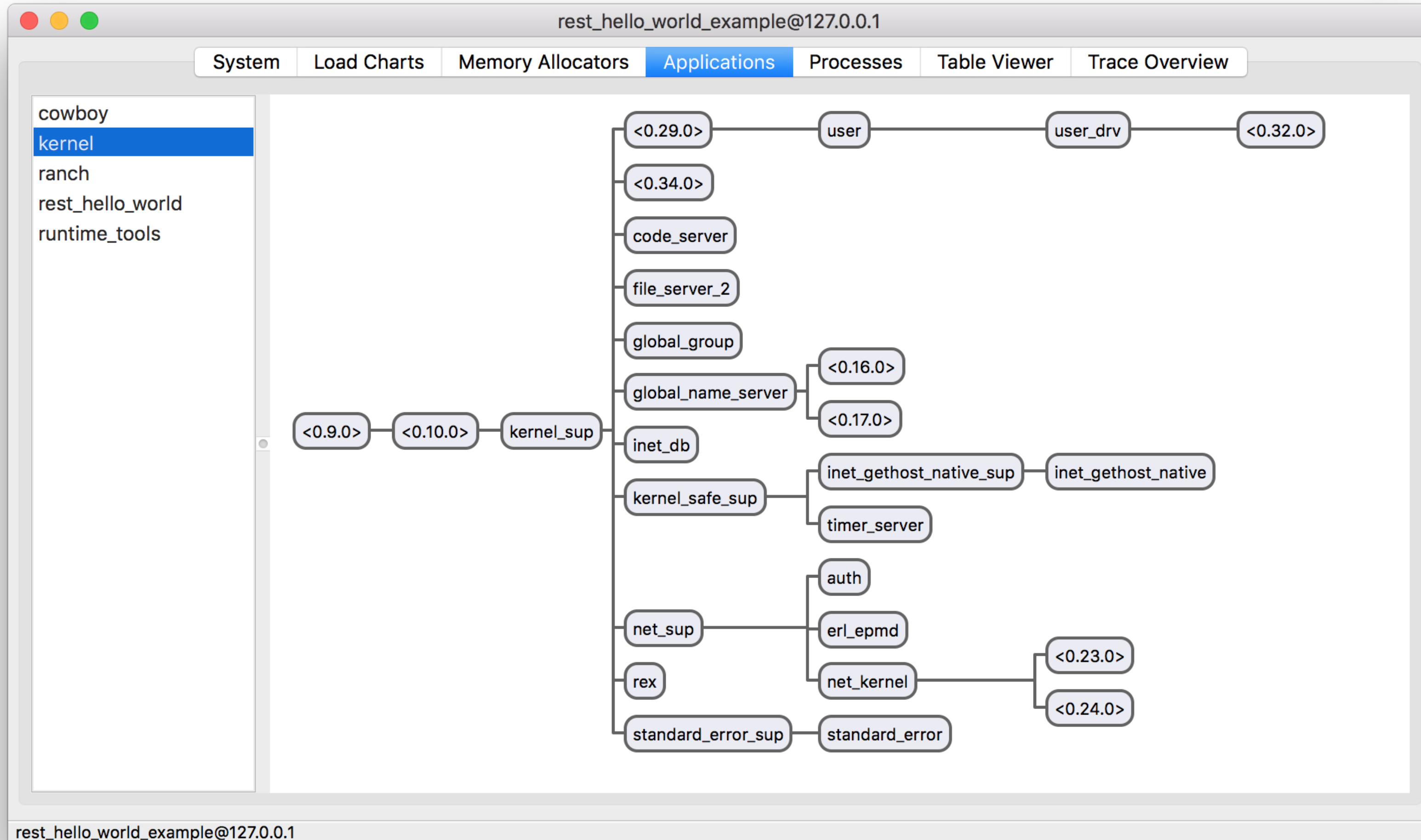
- max restarts
- max seconds

Pooly



Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

Observer



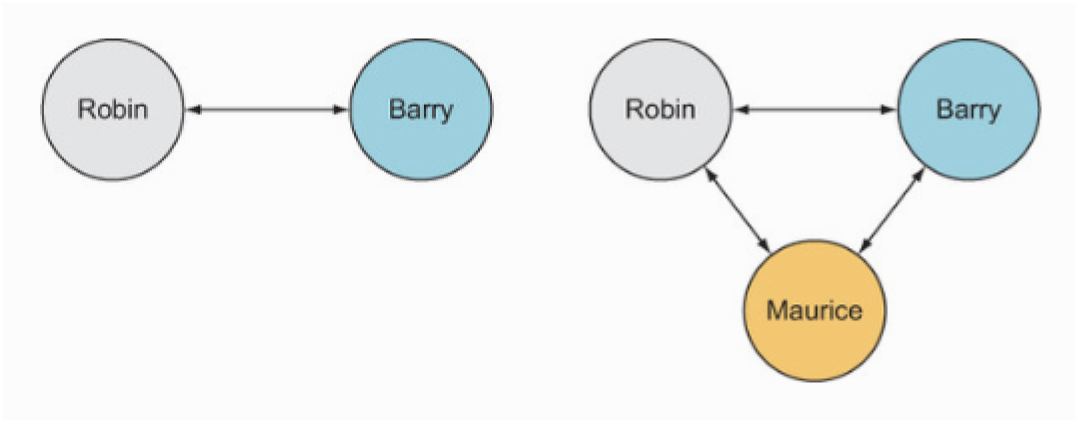
https://tkowal.wordpress.com/2016/04/23/observer-in-erlangelixir-release/

> One of the killer features of the Erlang VM is distribution—that is, the ability to have multiple Erlang runtimes talking to each other. Sure, you can probably do it in other languages and platforms, but most will cause you to lose faith in computers and humanity in general, just because they weren't built with distribution in mind.

Benjamin Tan Wei Hao. "The Little Elixir & OTP Guidebook."

Location transparent clusters!

Node connections are transitive

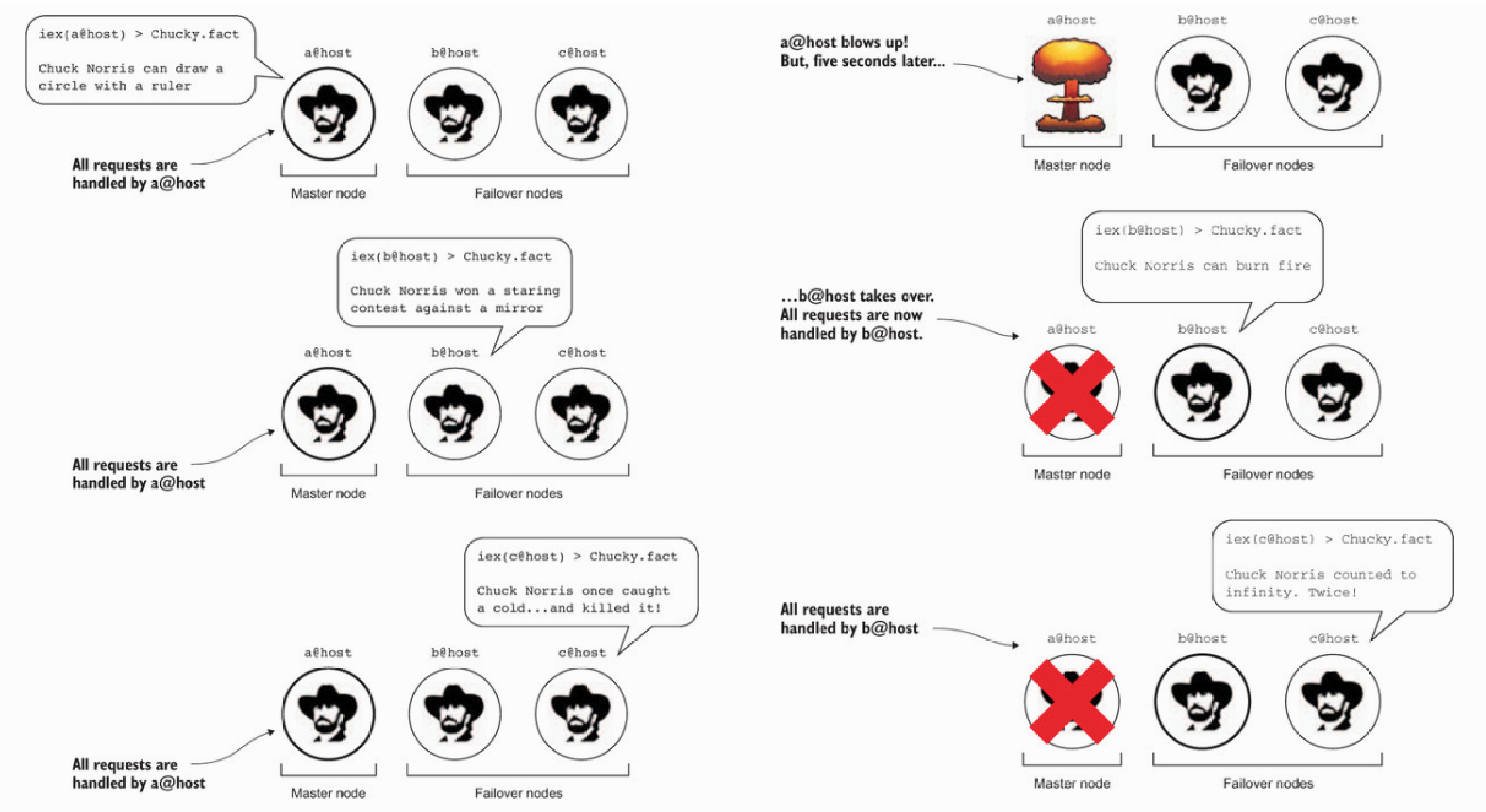


Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

Distribution & fault tolerance

- **Failover** - node crashes, another node takes over application
- **Takeover** - higher priority node takes over application

Chuck Norris



Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

GenStage

> GenStage is a new Elixir behaviour for exchanging events with back-pressure between Elixir processes.

http://elixir-lang.org/blog/2016/07/14/announcing-genstage/

GenStage

> Not only that, we want to provide developers interested in manipulating collections with a path to take their code from eager to lazy, to concurrent and then distributed.

http://elixir-lang.org/blog/2016/07/14/announcing-genstage/

Types

```java
public void onReceive(Object message) throws Exception {
  if (message instanceof String) {
    getSender().tell(message, getSelf());
  } else {
    unhandled(message);
  }
}
```

Akka Typed

http://doc.akka.io/docs/akka/current/scala/typed.html

Success typing (Dialyzer)

```
defmodule Cashy.Bug1 do

  def convert(:sgd, :usd, amount) do
    {:ok, amount * 0.70}
  end

  def run do
    convert(:sgd, :usd, :one_million_dollars)
  end

end
```

Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

Success typing (Dialyzer)

```
@spec convert(currency, currency, number) :: number
def convert(:sgd, :usd, amount) do
  amount * 0.70
end
```

Benjamin Tan Wei Hao. “The Little Elixir & OTP Guidebook.”

QuickCheck & Concuerror

The Little Elixir & OTP Guidebook

Benjamin Tan Wei Hao