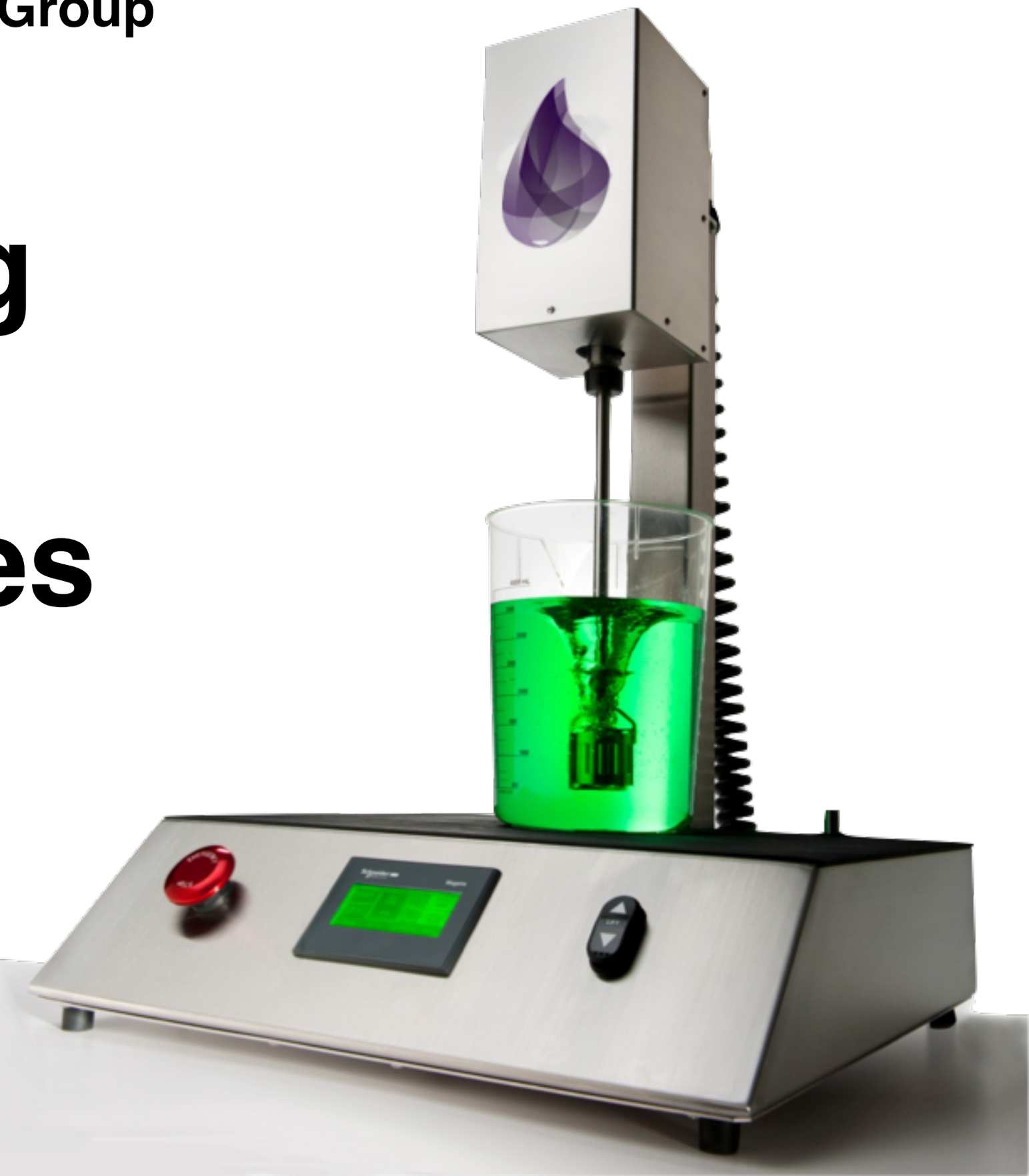
DevSpace **Elixir Study Group**
September 2014

Working with Processes

@xavierdefrang
github.com/xavier



Erlang is designed for **massive concurrency**.

Erlang processes are **light-weight** (grow and shrink dynamically) with **small memory footprint**, **fast to create and terminate** and the **scheduling overhead is low**.

http://www.erlang.org/doc/reference_manual/processes.html

RabbitMQ Management

localhost:15672/#/

Reader

RabbitMQ

User: guest
Cluster: rabbit@voigtkampff-2 (change)
RabbitMQ 3.3.1, Erlang R16B02

Log out

Overview | Connections | Channels | Exchanges | Queues | Admin | Visualiser

Overview

Totals

Queued messages (chart: last minute) (?)

1.0
0.0
08:40:10 08:40:20 08:40:3

Ready 0 msg
Unacknowledged 0 msg

Message rates (chart: last m

Currently idle

Global counts (?)

Connections: 0

Erlang processes	Memory
200	41MB
1048576 available	2.5GB high watermark

Nodes

Name	File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Uptime	Type
rabbit@localhost	28 256 available	3 138 available	200 1048576 available	41MB 2.5GB high watermark	21GB 48MB low watermark	2m 9s	Disc Stats *

Ports and contexts

Listening ports

Protocol	Bound to	Port
amqp	127.0.0.1	5672
clustering	::	25672

Display a menu

Erlang Processes

- are **not** OS processes or threads
- are **cheap** (~300 memory words)
- **scheduled** by the Erlang VM
- **all Erlang / Elixir code** executes in a process

Starting a Process

```
pid = spawn(fn -> 1 + 2 end)
#PID<0.43.0>
```

Process Identifiers

```
pid = spawn(fn -> 1 + 2 end)
#PID<0.43.0>

Process.alive?(pid)
false

Process.alive?(self())
true
```

Inter-Process Communication

- shared-nothing asynchronous message passing
- each process has a **mailbox**
- simple send / receive API built-in

Example

Process 1

```
…
send pid, {self, :sum, 1, 2}

…

receive do
  {^pid, :ok, sum}
    IO.puts sum
end
…
```

Process 2

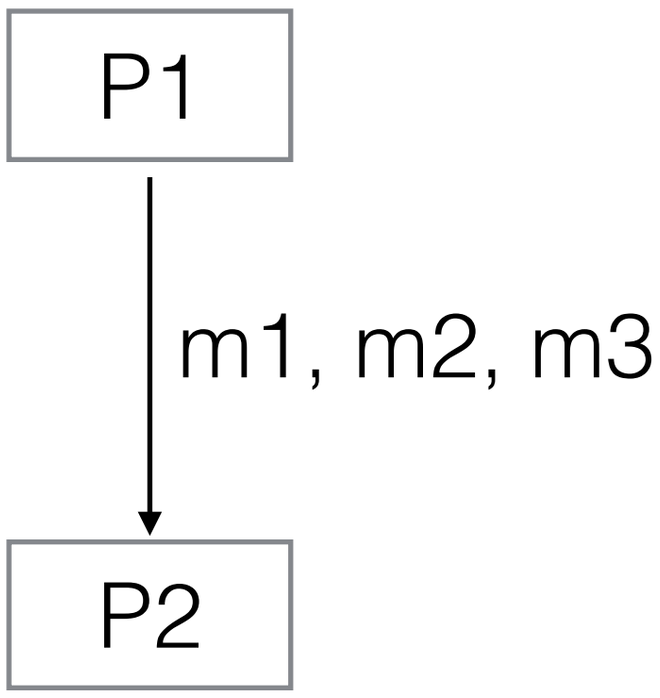
```
…
receive do
  {sender, :sum, a, b} ->
    send sender, {self, :ok, a+b}
  {sender, :subtract, a, b} ->
    send sender, {self, :ok, a-b}
  …
end
…
```

Receiving Messages

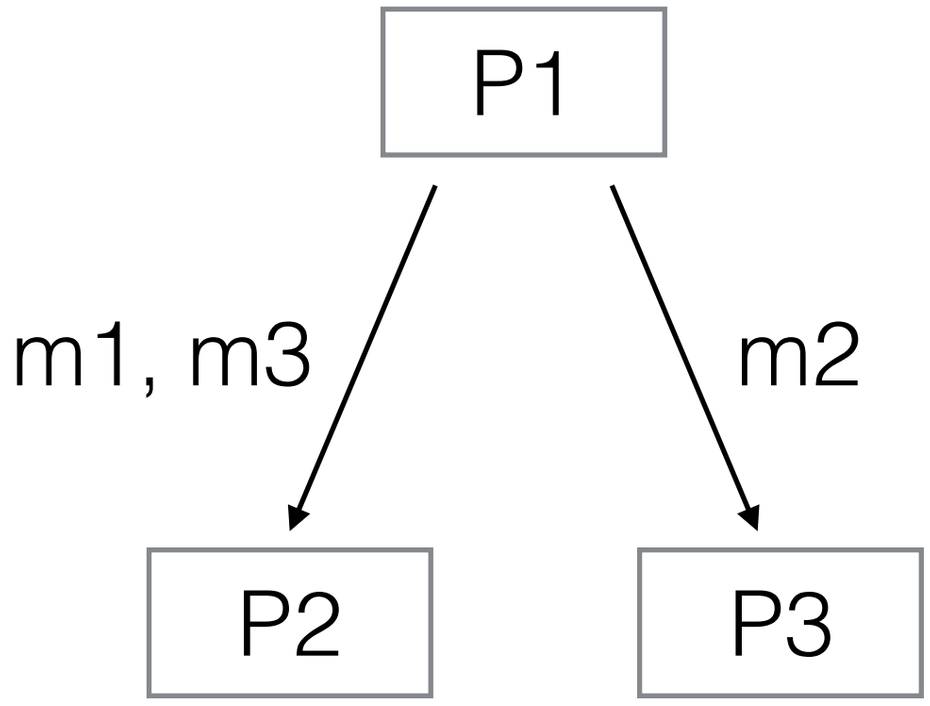
- `receive` **blocks** until a new message matches any patterns
- unmatched messages **remain in the mailbox**

Message Order

Example: message send sequence m1, m2, m3



Message order is guaranteed **per process**



No guarantee that m2 will be received after m1 and before m2

Timing Out

```
receive do
  {:hello, msg}  -> msg
after
  1_000 -> "timed out after 1s"
end
```

Handling Failures

```
spawn fn -> raise "oops" end
#PID<0.58.0>
```

Linking Processes

```
spawn_link fn -> raise "oops" end
#PID<0.60.0>
** (EXIT from #PID<0.41.0>) an exception was raised:
    ** (RuntimeError) oops
        :erlang.apply/2
```

Linking can also be done manually using `Process.link/1`

Named Processes

```
Process.register(pid, :my_proc)
# true
send :my_proc, {:msg, self}
# …
Process.registered
# [:elixir_sup, :error_logger, …
    :my_proc]
Process.whereis :my_proc
# #PID<0.68.0>
```

Managing State

- Processes as key mechanism of **state management**
- Encapsulation of “persistent” state
- Mutations as response to incoming messages
- Sounds a lot like OOP, doesn’t it?

State is the **value** of an **identity**
at one point in **time**

Rich Hickey

Example: Counter

```
c1 = Counter.new
c2 = Counter.new

Counter.inc(c1)
Counter.inc(c2)

IO.inspect Counter.value(c1)
# 2
IO.inspect Counter.value(c2)
# 1
```

Counter Initialization

```elixir
defmodule Counter do
  def new do
    spawn_link(fn -> loop(0) end)
  end

  # …

end
```

Counter Loop

```
defp loop(counter) do
  receive do
    {:inc, _} ->
      loop(counter + 1)
    {:val, sender} ->
      send sender, {:ok, self(), counter}
      loop(counter)
  end
end
```

Incrementing

```
def inc(counter) do
  send counter, {:inc, self}
end
```

Querying the Value

```
def value(counter) do
  send counter, {:val, self}
  receive do
    {:ok, ^counter, val} ->
      val
  end
end
```

⚠️ Caret Operator

```
iex(1)> a = 1
# 1
iex(2)> ^a = 1
# 1
iex(3)> a = 2
# 2
iex(4)> ^a = 1
** (MatchError) no match of
   right hand side value: 1
```

The caret indicates that we want to match against the value currently bound to the variable rather than rebinding it

Nodes

- Nodes are **named Erlang runtimes**
- Code can run **transparently** on the same or a different host
- **Authentication** using a **cookie** (shared-secret)
- Node connections are:
  - **symmetric**
    if A connects to B then B is also connected to A
  - **transitive** (by default)
    if A connects to B and B is connected to C than A is connected to C

Starting a Node

```
$ iex --name node_name --cookie secret
```

```
$ elixir --name node_name --cookie secret

...
```

Host 1

```
[host1] $ iex --name alice --cookie secret
```

```
iex(alice@host1)1> Node.list
[]
```

```
iex(alice@host1)2> Node.list
[:"bob@host2"]
```

Host 2

```
[host2] $ iex --name bob --cookie secret
```

```
iex(bob@host2)1> Node.connect(:"alice@host1")
true
```

```
iex(bob@host2)2> print_node_name =
  fn -> IO.puts Node.self end
#Function<20.90072148/0 in :erl_eval.expr/5>
```

```
iex(bob@host2)3>
  Node.spawn(:"alice@host1", print_node_name)
alice@host1
#PID<9088.64.0>
```

Hold on a second...

```
iex(bob@host2)2> print_node_name = fn -> IO.puts Node.self end
iex(bob@host2)3> Node.spawn(:”alice@host1”, print_node_name)
bob@host1
```

The code of `print_node_name` is executed on **host1**
but `IO.puts` writes to **host2** standard output

`IO.puts` actually communicates
with an **I/O server** running on **host2**

Code created on a given node inherits
its **process hierarchy**

Useful Abstractions

- Processes are rarely used as-is
- Larger software is designed as a tree of supervised applications using the **OTP API** (gen_server, gen_fsm, gen_event, supervisor, …)
- The Elixir stdlib provides a handful of ready-to-use convenient abstractions for common use cases:
  - **Agent**, for state management
  - **Task**, for asynchronous workers

Further Readings

- Elixir Process module
  http://elixir-lang.org/docs/stable/elixir/Process.html

- Elixir Node module
  http://elixir-lang.org/docs/stable/elixir/Node.html

- How Erlang does scheduling?
  http://jlouisramblings.blogspot.be/2013/01/how-erlang-does-scheduling.html

- Understanding the Erlang Scheduler
  https://www.youtube.com/watch?v=tBAM_N9qPno

Playing Around

- In **IEx**:
  - **First:**
    - Send some messages to the **current process**
    - Receive and print those messages
- **Then:**
  - Spawn a **new process** performing a simple calculation
  - Retrieve the result of the calculation in the IEx shell

Exercises

https://github.com/belgian-elixir-study-group/meetup-materials

- In `meetup2`:

1. Counter
2. Flush
3. Parallel Map

Use cookie devspace