Message Passing, Erlang, and Elixir

History of Erlang

- Erlang either refers to Danish mathematician and engineer Agner Krarup Erlang, or alternatively, as an abbrev. of "Ericsson Language“.
- Experiments with Erlang started in Ellemtel Computer Science Laboratory in 1987.
- First version was developed by Joe Armstrong.

Agner Krarup Erlang (1878-1929)
Inventor of the fields: traffic engineering and queueing theory (foundation of telecom network studies).

Cool features of Erlang

- Started out as a concurrent Prolog.
- High reliability. (e.g. Ericsson Switch)
  - "You need mechanisms and primitives provided to the general programmer or the developer in order for the systems to have these properties, otherwise they tend to get lost in the way."
    – Lennart Ohman, Developer of Open Telecom Platform (OTP), MD of Sjoland & Thyselius Telecom AB in Sweden.
- Seamless scaling to large number of processes (literally 100,000 processes) through message passing communication.
- Functional programming (Seq.) + OO (Conc.).
- Concurrency for the sake of modularity, reliability etc.
- Library support (web-server, databases etc.)

Properties

- Functional
- Dynamic typing
- Concurrency-oriented programming
  - Asynchronous message passing
  - Lightweight processes (100’s of thousands!)
  - Not quite like threads…
- Anonymous functions
- List comprehensions, map, fold, …
- Tuples, lists, binaries, …
- …

Projects using Erlang

- Ericsson AXD301 switch (1998).
  - reported to achieve a reliability of nine “9”s (i.e. 1 sec of downtime in 1 billion seconds, roughly 30 years.)
- CouchDB, a document based database that uses MapReduce.
- ejabberd, instant messaging server
  - Facebook chat system based on ejabberd.
- Twitterfall, a service to view trends and patterns from Twitter
- whatsApp, a messaging service
- …

The Future of Erlang

Joe Armstrong, author of Erlang

Ralph Johnson, co-author of the now-legendary book, "Design Patterns"

- Scalability: "Virtually all language use shared state concurrency. This is very difficult and leads to terrible problems when you handle failure and scale up the system...*Some pretty fast-moving startups in the financial world have latched onto Erlang*; for example, the Swedish www.kreditor.se." – Joe Armstrong, "Programming Erlang".

- Promise for multi-core apps: "I do not believe that other languages can catch up with Erlang anytime soon. It will be easy for them to add language features to be like Erlang. It will take a long time for them to build such a high-quality VM and the mature libraries for concurrency and reliability. So, Erlang is poised for success. *If you want to build a multicore application in the next few years, you should look at Erlang.*" – Ralph Johnson (UIUC), "Erlang, the next Java".

Erlang is cool, but it uuugly…

-David Hart, former CS 330 TA

Step in Elixir…

- Elixir was designed with excellent and accessible tooling, documentation, and resources in mind. On the more technical side, Elixir provides a macro system and polymorphism via protocols, both aiming to make the language more extensible.

  – Jose Valim, Creator of Elixir

Elixir functions

```
defmodule Recurse do

    def fac(0) do 1 end
    def fac(n) do
        n*fac(n-1)
    end

end
```

Elixir: pattern matching

- Pattern-matching selects components of the data
- ‘_’ is a “don’t care” pattern (not a variable)
- ‘[]’ is the empty list
- ‘[x,y,z]’ is a list with exactly three elements
- ‘[x,y,z|tail]’ has three or more elements

Elixir functions

```
defmodule MyLists do

    def reverse(l) do reverse(l,[]) end

    def reverse([h|t], l) do
        reverse(t, [h|l])
    end

    def reverse([], l) do l end
end
```

Symbols and tuples

```
defmodule Tut3 do

    def convert_length({:centimeter, x}) do
        {:inch, x/2.54}
    end

    def convert_length({:inch, y}) do
        {:centimeter, y*2.54}
    end
end
```

Guards

```
defmodule Tut6 do

        def list_max([head|rest]) do list_max(rest,head) end

        def list_max([], result) do result end

        def list_max([head|rest], result_so_far) when head > result_so_far do
                list_max(rest, head)
        end

        def list_max([head|rest], result_so_far) do
                list_max(rest, result_so_far)
        end
end
```

Notes

- Can only assign to variables once (with “=“)
- Also, “===“ means “check if equal to”
- Modules start with capital letters
- Variables and functions start with lower-case letters
- Symbols start with a colon, :

List comprehensions

```
> for n <- [1,2,3,4] do 2*n end

[2,4,6,8]

> for n <- [1,2,3,4,5,6,7,8,9,10], n < 5 do n*n end

[1,4,9,16]
```

Note: These are not the common notion of “for loops”.

Quicksort

```
defmodule QSort do

    def qsort([]) do [] end
    def qsort([pivot|rest]) do
        smaller = for n <- rest, n < pivot do n end
        larger = for n <- rest, n >= pivot do n end
        qsort(smaller) ++ [pivot] ++ qsort(larger)
    end
end
```

++ concatenates two lists; -- removes items from a list

Quicksort with Enum

```
defmodule SimpleQSort do

        def qsort([]) do [] end
        def qsort([pivot|rest]) do
                {smaller, larger} = Enum.split_with(rest,fn(x) -> x < pivot)
                qsort(smaller) ++ [pivot] ++ qsort(larger)
        end
end
```

Concurrency oriented programming

- Processes are totally independent
  - imagine they run on different machines
- Process semantics = No sharing of data = Copy-everything message passing.
  - Sharing = inefficient (can’t go parallel) + complicated (mutexes, locks, ..)
- Each process has an unforgeable name
- If you know the name of a process you can send it a message
- Message passing is ”send and pray”
  - you send the message and pray it gets there
- You can monitor a remote process

COPL

- A language is a COPL if:
  - Process are truly independent
  - No penalty for massive parallelism
  - No unavoidable penalty for distribution
  - Concurrent behavior of program same on all OSs
  - Can deal with failure

Simple server and client

```
defmodule Christmas do
	def northpole() do
		receive do
			{message, pid} ->
				send(pid,{"I Amazon Primed it for you. - Santa",self()})
		end
		northpole()
	end

	def writeLetter(pid,message) do
		send(pid, {message, self()})
		waitByMailBox()
	end

	def waitByMailBox() do
		receive do
			#Open Letter
			{message, pid} -> IO.puts(message)
		end
	end
end
```

Console Output

```
$iex
Erlang/OTP 20 [erts-9.2.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-poll:false] [dtrace]
Interactive Elixir (1.6.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c("Christmas.exs")
[Christmas]
iex(2)> pid = spawn &Christmas.northpole/0
#PID<0.91.0>
iex(3)> Christmas.writeLetter(pid,"Santa, please bring me a teddy bear.")
I Amazon Primed it for you. - Santa
:ok
iex(4)>
```

Spawning functions, receive, and send

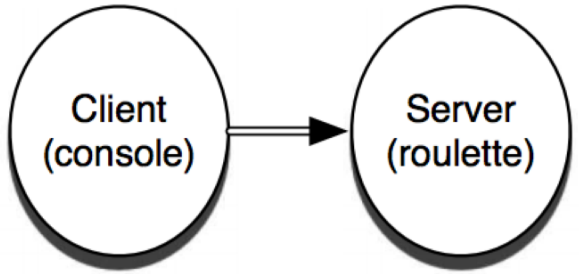
- Functions can be spawned and assigned a Proccessor Identifier (PID).
- The syntax looks like pid = spawn &ModuleName.function/arity
- Erlang and Elixir always specifies the arity of their functions.
- Examples: northpole/0, writeLetter/2, quicksort/1

- **receive** and **send** are the primary way to communicate between spawned processes.
- **Beware!** **receive** is a blocking call.

Christmas Extended

```
defmodule ChristmasExt do

        def northpole() do
                receive do
                        {message, age, pid} -> respond(message, age, pid)
                end
                northpole()
        end

        def respond(message, age, pid) when age >= 18 do
                send(pid,{"No, you are an adult, stop writing me. - Santa",self()})
        end

        #The year Santa lost the naughty list...
        def respond(message, age, pid) do
                if String.contains?(message, "good") do
                        send(pid,{"Sure thing, expect it Christmas Eve. - Santa",self()})
                else
                        send(pid,{"Sorry, you are on the naughty list. - Santa",self()})
                end
        end
```

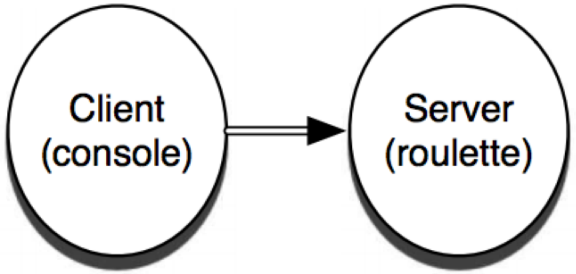
Process Exiting



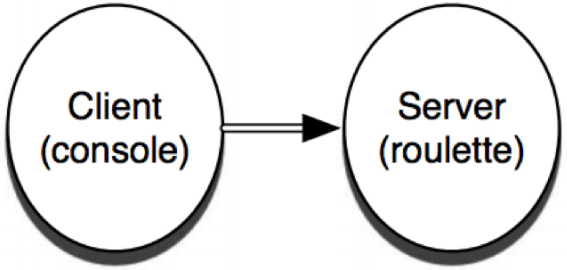

```
defmodule Roulette do

        def gun() do
                receive do
                        {5,_} ->
                                IO.puts("Bang!!!")
                                Process.exit(self(),:kill)
                        {_,_} -> IO.puts("click...")
                end
                gun()
        end

        def pullTrigger(gun_pid) do
                chamber = Enum.random(0..6)
                send(gun_pid,{chamber,self()})
        end
end
```

Console Output

iex(51)> pid = spawn &Roulette.gun/0
#PID<0.146.0>
iex(52)> Roulette.pullTrigger(pid)
click...
{6, #PID<0.79.0>}
iex(53)> Roulette.pullTrigger(pid)
click...
{3, #PID<0.79.0>}
iex(54)> Roulette.pullTrigger(pid)
Bang!!!
{5, #PID<0.79.0>}
iex(55)> Roulette.pullTrigger(pid)
{4, #PID<0.79.0>}
iex(56)> Roulette.pullTrigger(pid)
{0, #PID<0.79.0>}



Process Restarting



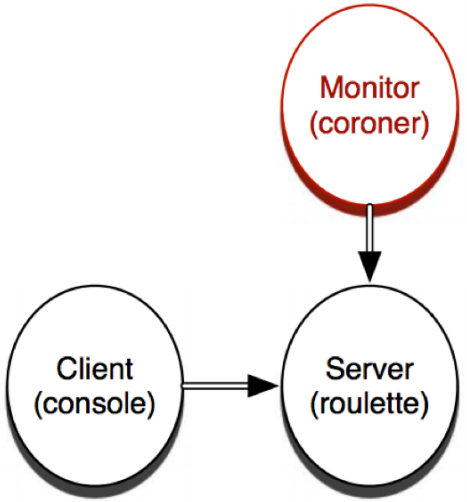

```
defmodule RouletteExt do

      def gun() do ....
      def pullTrigger(gun_pid) do ...

      def medic(pid) do
            if not Process.alive?(pid) do
                  IO.puts("Reviving...")
                  pid = spawn &RouletteExt.gun/0
            end
            pid
      end
end
```

Console Output

```
iex(58)> pid = spawn &RouletteExt.gun/0
#PID<0.155.0>
iex(59)> RouletteExt.pullTrigger(pid)
click...
{2, #PID<0.79.0>}
iex(60)> RouletteExt.pullTrigger(pid)
click...
{4, #PID<0.79.0>}
iex(62)> RouletteExt.pullTrigger(pid)
Bang!!!
{5, #PID<0.79.0>}
iex(63)> pid = RouletteExt.medic(pid)
Reviving...
#PID<0.161.0>
iex(64)> RouletteExt.pullTrigger(pid)
click...
{6, #PID<0.79.0>}
```

Process Monitoring

```
defmodule RouletteAuto do

    def gun() do ....
    def pullTrigger(gun_pid) do ...

    def medic(pid) do
        Process.monitor(pid)
        receive do
            _ ->
                IO.puts("Reviving...")
                pid = spawn &RouletteExt.gun/0
        end
        medic(pid)
    end
end
```



There’s a better way to do this, but more on that later…

Erlang: “Let it crash!”

“The view is that you don't need to program defensively. If there are any errors, the process is automatically terminated, and this is reported to any processes that were monitoring the crashed process. In fact, defensive programming in Erlang is frowned upon.”

Erlang: “Let it crash!”

“Only program the happy case, what the specification says the task is supposed to do.”

“When writing code from a specification, the specification says what the code is supposed to do, it does not tell you what you’re supposed to do if the real world situation deviates from the specification”

“So what do the programmers do? ... they take ad hoc decisions”

Erlang: “Let it crash!”

“If a hardware failure requires any immediate administrative action, the service simply won’t scale cost-effectively and reliably. The entire service must be capable of surviving failure without human administrative interaction. Failure recovery must be a very simple path and that path must be tested frequently.

Armando Fox of Stanford has argued that the best way to test the failure path is never to shut the service down normally. Just hard-fail it. This sounds counter-intuitive, but if the failure paths aren’t frequently used, they won’t work when needed.”

Erlang: “Let it crash!”

- Do I know how I’m supposed to handle an error here?
- If not, then should I handle it? (Thus going back to specification)
- If something goes wrong in this function call, **can** I continue?
  - E.g. Line 2 depends on a file descriptor from line 1
- When I restart, can I **recover** from the crash here?
  - If not then it is a good indication that you need a specification for how to handle the error. Can I reset my state? Do I need to clean up? etc are also good questions to ask yourself
- Am I crashing in a valid place?
  - E.g. you should never crash (intentionally) inside a gen_server unless it is some kind of a worker process. A “main” process shouldn’t really be allowed to crash in the same sense as a worker process can. A worker process, say for an HTTP request, shouldn’t really be very defensive (perhaps a top-level try-catch to return some useful error)