Large-Scale Computing in Erlang

Stephen Franklin, Brook Tamir, Gregory Chang

Agenda

- Where Erlang is Used
- History
- Language Features
- Basics
- Distributed Computing

Where Erlang is Used

- Whatsapp
- Goldman Sachs
- BT Mobile
- Discord
- Ericsson
- Cisco
- Nintendo

History

Erlang, while also being a syllabic abbreviation of "Ericsson Language", is named after Danish mathematician and engineer Agner Krarup Erlang.

- Erlang was developed at the Ericsson and Ellemtel Computer Science Laboratories.
- Erlang was created as an experiment to see if declarative programming techniques could be applied to large industrial scale telecom switching systems that needed to be incredibly reliable and scalable.
- Scientists at Ericsson realized that many of the problems related to telecommunications could also be applied to a wide variety of real-time control problems faced in other industries and released it as a general purpose language.

Source: https://erlang.org/download/erlang-book-part1.pdf

Language Features

Erlang is a declarative, general purpose, functional programming language, built with concurrency in mind.

Main features:

- Declarative
- Concurrent
- Real-time (Soft)
- Continuous operation
- Robust
- VM/Real-time Garbage Collected
- No shared memory
- Easily Integrate with programs written in other languages.
- Hot Swapping

Source: https://erlang.org/download/erlang-book-part1.pdf

Basics

Erlang is primarily a functional programming language.

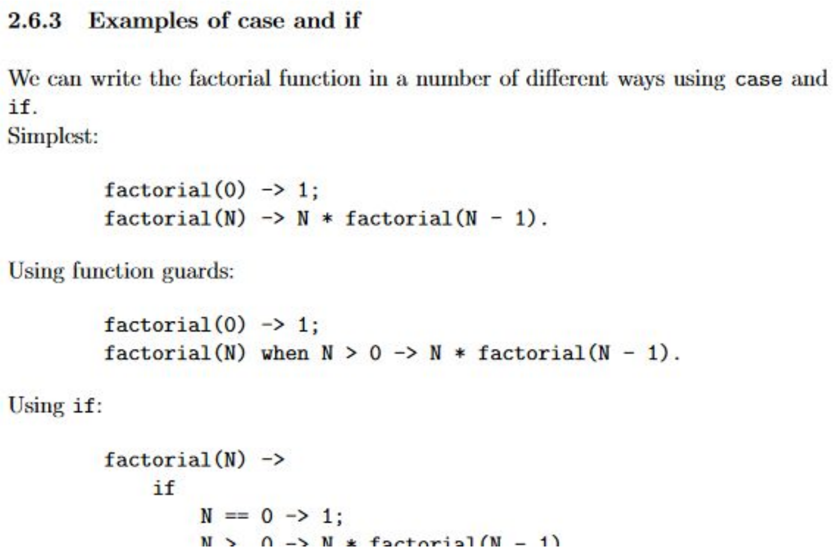
A core difference between Erlang and an imperative language like Java is that there is a heavy focus on processes.

Variables are immutable.

Individual blocks of code produces consistent output values.

Basic program syntax and execution

- Since the language employs pattern matching the Erlang VM will decide which function to employ based pattern matching of the parameters.
- If **factorial(4)** will match to **factorial(N) -> N * factorial(N - 1)**
- Next, **factorial(3)** will match to **factorial(N) -> N * factorial(N - 1)**
- Next, **factorial(2)** will match to **factorial(N) -> N * factorial(N - 1)**
- Next, **factorial(1)** will match to **factorial(N) -> N * factorial(N - 1)**
- Finally, **factorial(0)** will match to **factorial(0) -> 1**



2.6.3 Examples of case and if

We can write the factorial function in a number of different ways using case and if.
Simplest:

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

Using function guards:

```
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

Using if:

```
factorial(N) ->
    if
        N == 0 -> 1;
        N > 0 -> N * factorial(N - 1)
```

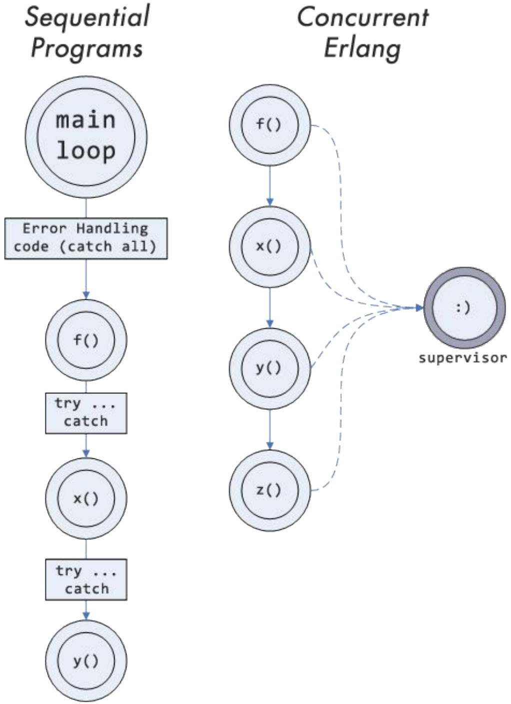
Program/Diagram Source: https://erlang.org/download/erlang-book-part1.pdf

Erlang Error Handling "Let it crash"

- Instead of burdening yourself with defensive programming principles there is a philosophy "let it crash".

- Since each piece of the application is broken out into small processes. The supervisor will monitor child processes and is responsible for managing them.

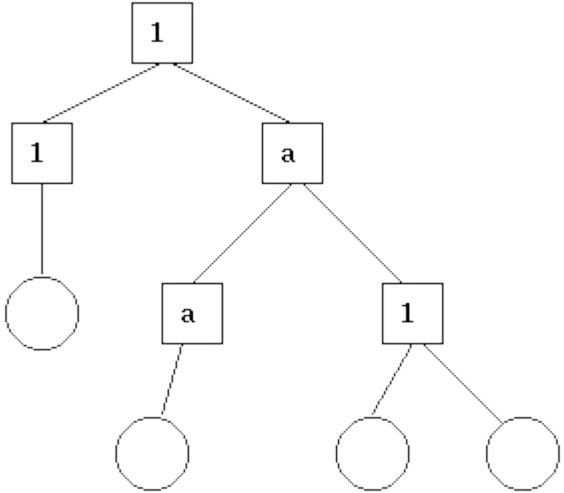
- If a child process crashes, the supervisor will start, stop, or restart all the other processes it supervises depending on the selected restart strategy.

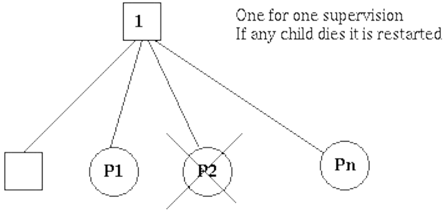


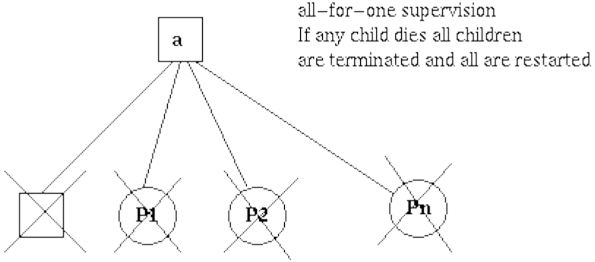
Sources
Diagram: https://ferd.ca/an-open-letter-to-the-erlang-beginner-or-onlooker.html
Content: https://erlang.org/doc/man/supervisor.html

Supervisor Trees

- Workers are the actual processes that perform computation. (circles)
- Supervisors monitor workers and can decide what to do when a child process exits. (Squares)
- Supervisors can monitor other supervisors.





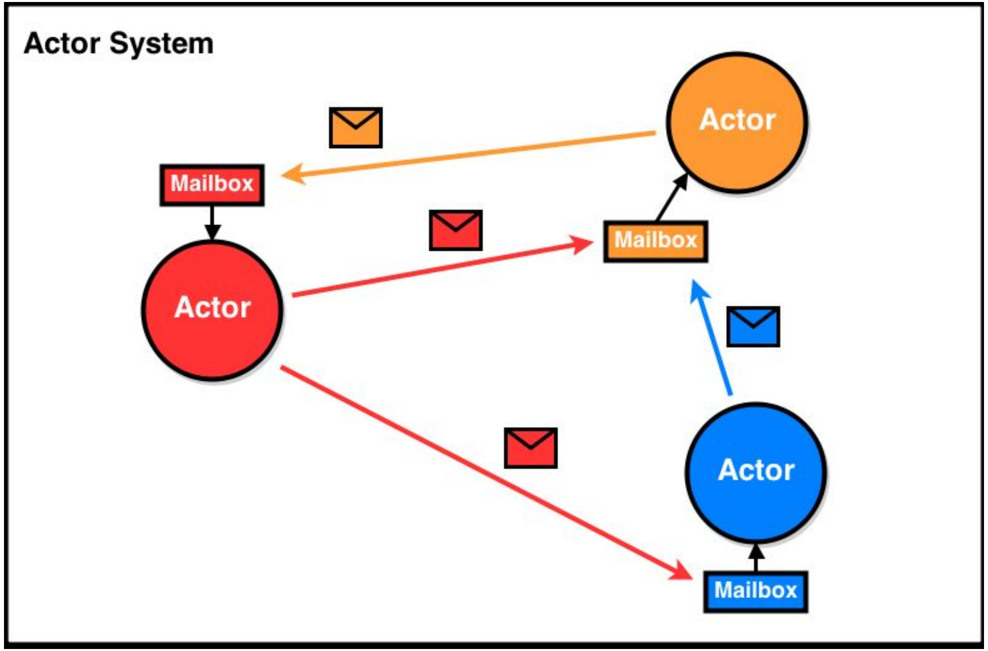

a
all–for–one supervision
If any child dies all children
are terminated and all are restarted
P1
P2
Pn

Source for content and diagrams: http://erlang.org/documentation/doc-4.9.1/doc/design_principles/sup_princ.html

Erlang Concurrency Model

“The philosophy behind Erlang and its concurrency model is best described by Joe Armstrong’s tenets:

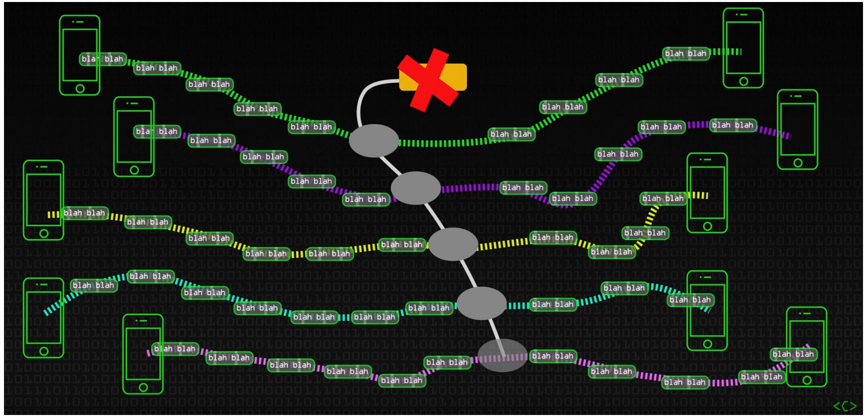
- The world is concurrent.
- Things in the world don’t share data.
- Things communicate with messages.
- Things fail.”



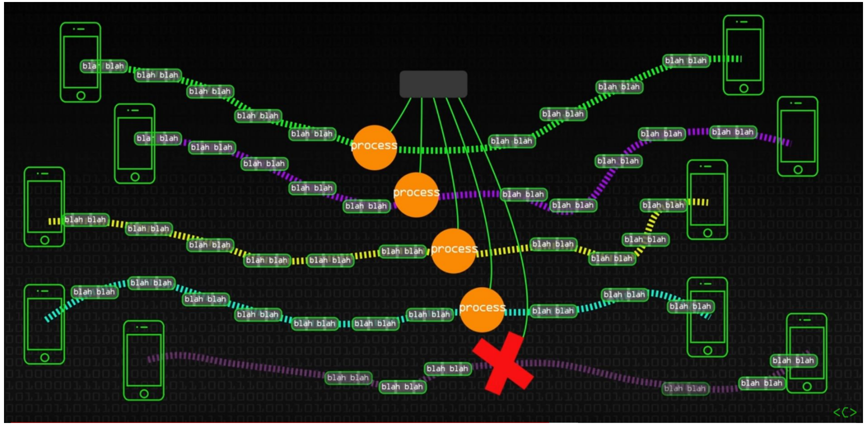
Source:
https://www.oreilly.com/library/view/erlang-programming/9780596803940/ch04.html
Diagram:
https://blog.scottlogic.com/2014/08/15/using-akka-and-scala-to-render-a-mandelbrot-set.html

Erlang vs. Thread Concurrency Models

Traditional, thread-based, concurrency model encounters bug that causes fatal error in process.



Erlang process-based, concurrency model encounters bug that causes fatal error in process.



Both Diagrams from the fantastic Computerphile Youtube video: 'Erlang Programming Language - Computerphile'
https://youtu.be/SOqQVoVai6s?t=603

Distributed Computing

- Many instances of a server rather than a single server
- Hot swapping (live code reload)
- Fault tolerance lead naturally to scalability
- Dataflow impacted by physical architecture

Discussion Questions!

1. Would you use Erlang if you were working on distributed applications?

2. Should we let the rarity of a programming language like Erlang dictate our decisions about whether or not to use it? E.g. if Erlang is the best choice for the backend of a startup, might it still not be the best choice?

3. Are there any cons of the “Let it crash” philosophy that Erlang employs?