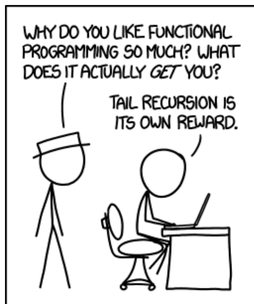

WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY GET YOU?
TAIL RECURSION IS ITS OWN REWARD.

Fault-tolerance with Erlang

Seminar on Scientific Computing Languages 2013/14

Jan Winkelmann

January 9, 2014

Table of contents

1. Introduction
2. Erlang Actors
3. Fault-tolerant Applications in Erlang
4. Conclusion

Section 1

Introduction

Erlang

- Developed by Ericsson in 1987 for telephony routers
- Requirements:
- Soft real-time capable
- Provide high availability
  (less than 5 seconds downtime per year)
- Provide fault-tolerance

```
Hi = fun() -> io:format("Hello World~n") end. Hi().
```

Erlang: Facts

- Strongly and dynamically typed (like Ruby, Python)
- Impure functional, compiled (like Scala)
- Actor based
- Concurrent
- Extremly lightweight processes in Virtual Machine
- Out-of-the-box distributed computing

Features I will not talk about

- High availability (i.e. hot code loading)
- Real-time programming
- Distributed Erlang

We will focus on Erlangs fault-tolerance features

Why design fault-tolerant applications?

- Your program will not fail when errors occur
- Modern languages have features to deal with some errors
  Exceptions, anyone?
- Consider temporary hardware faults
- Problem is even worse for distributed programming
- Erlang has a different approach to these problems

- Trends in HPC: Exascale Computing
  - Traditional programming approach becomes difficult
  - Mean time to failure will increase
  - MPI has no build-in fault-tolerance

Section 2

Erlang Actors

What are Actors

- Model for concurrent computation from theoretical CS
- Actors can:
  - Compute,
  - Send/receive messages, and
  - Spawn new actors.
- Used in a few languages (Scala, Erlang, ...)

Actors in Erlang

- Processes in Erlang are actors
- No shared memory, very little shared state
- Leightweight:

  Extremely fast creation, destruction, and scheduling

Messages in Erlang

- Communication via message passing
- Messages sent to existent processes will arrive
- But receiving process might
  - Not exist
  - Crash before reading message
  - Crash before replying
- No errors for all of these

Actors in Erlang: Example

```
-module(helloworld).
-export([start/0, server/0]).

start() ->
    spawn(helloworld,server,[]).
server() ->
    receive
        {Msg, Client} ->
            erlang:display(Msg),
            Client ! sucess,
            server();
        _ ->
            io:format("Stopping~n");
    end.
```

Actors in Erlang: Example

```
8> c(helloworld).
9> Server = helloworld:start().
10> Server ! {12, self()}.
    12
11> flush().
    Shell got sucess
12> Server ! {notvalid}.
    Stopping
13> Server ! {12, self()}.
```

From here on we need Actors

- Questions about Actors?

Section 3

Fault-tolerant Applications in Erlang

Reminder: Motivation

- Let's talk about fault-tolerance (in distributed systems):
- Best example: temporary faults, hardware failure
- What to do when something crashes?
- Error handling can be very complex

Let it fail!

- Design Pattern: **Let it fail!**
- Recover only if easily possible
- Fail as early as possible,
- then retry

Actor based fault-tolerance

Let it fail for actors:

- If error recovery not easily possible: fail
  i.e. via exception
- Dependent processes also fail
- Restart all failed processes, retry

Make process fail	✓
Make dependent processes fail	✗
Notice failed processes	✗

Link: Dependencies between Processes

- Link: logical connection between two processes
- Used for expressing dependency among processes
- Established via function call
- If process dies it sends an *exit signal* to all linked processes
- Processes fail when receiving an exit signal

Process fail	✓
Dependent process fail	✓
Notice failed processes	✗

Linking: Example

```
1> self(). %Pid of shell
<0.32.0>
2> link(spawn(fun() - > %Exit with error after 5s
    timer:sleep(5000), exit(reason) end)).
true
...
** exception error: reason
3> self(). %Pid of shell
<0.37.0>
```

The shell crashed, because the linked actor exited abnormally

Linking: Respawning failed processes

- Normally linked processes die when receiving exit signal
- Processes can choose to trap exit signals
- Exit signal is then added to the process' inbox.
- Can be used to respawn failed processes
- Worker/Supervisor Design Pattern

Process fail	✓
Dependent process fail	✓
Notice failed processes	✓

Restart Strategies

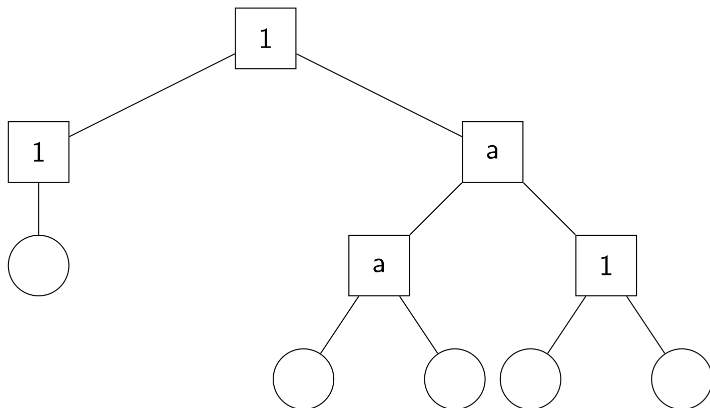
What does a supervisor do when a watched process fails?

- One for one
  Restart only failed process
- One for all
  Equivalent to linking watched processes
  Restart all watched processes if one process fails

Supervisor Trees

- Supervisors restart watched processes,
- according to a restart strategy.
- Simplification: Each process is watched by only one supervisor
- Allow supervisors to watch supervisors
- $\Rightarrow$ Supervisor trees

Supervisor Trees: Example



1: One for one restart strategy; a: One for all restart strategy

Wrap-up: Fault-tolerant parallel Erlang

- Erlang can work distributed out of the box
- Supervisors may be on separate node
- Restart children on another node

Erlang/OTP: Batteries included

Open Telecom Platform comes with

- Supervisor trees,
- Finite state machines,
- Gen_server (sychronous messaging)
- A key value store (Mnesia)

Section 4

Conclusion

Summary

- Erlang is functional and actor based
- Fast, distributed, process spawning
- Express dependencies between processes
- Restart of failed processes
- $\Rightarrow$ Supervisor trees
- Remark: not tolerant against all possible faults

Erlang and Scientific Computing

Use Erlang for scientific computing with these requirements:

- Fault-tolerant,
- High Availability,
- Massively Large Scale,
- Soft real-time

Example: Disco

- Disco, a MapReduce framework by Nokia
- `http://discoproject.org/`
- Basically a Hadoop competitor

- Also: Neural Networks

- ejabberd, CouchDB, ZeroMQ, Wings3D, WhatsApp, High-frequency trading, ...

Sources

- Programming Erlang by Joe Armstrong
  ISBN: 978-1937785536
- Seven Languages in Seven Weeks by Bruce Tate
  ISBN: 978-1934356593
- Erlang Documentation: `erlang.org/doc`
- `http://learnyousomeerlang.com`

Further Resources

- Erlang
  - Seven Languages in Seven Weeks by Bruce Tate
    ISBN: 978-1934356593
  - learnyousomeerlang.com
- Functional Programming:
  - learnyouahaskell.com
  - Real World Haskell by Bryan O'Sullivan
    ISBN: 978-0596514983 book.realworldhaskell.org/read
- Take a look look at elixir-lang.org a "functional, meta-programming aware language built on top of the Erlang VM."

The End.

Questions?

There are a lot of things missing

Ask me about:

- Functional Programming
- Types
- Toolchain (compiler, debugger)
- C language bindings and Cuda
- Parallelism vs. Concurrency
- High performance Erlang, HiPE
- Handling binary data

Factorial

```
-module(factorial).
-export([factorial/1]).

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

Factorial: DEMO

```
1> c(factorial).
{ok,factorial}
2> factorial:factorial(100).
93326215443944152681699238856266700490715968264381621468592963
```