FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Konzepte moderner Programmiersprachen (KmPS)

(Wahlfach, Modulnummer 55685)

Wintersemester 2025/2026

Prof. Dr. Andreas Claßen
(zusammen mit Prof. Dr. Heinrich Faßbender)
Fachbereich 5 Elektrotechnik und Informationstechnik
FH Aachen

Nebenläufigkeit & Parallelismus in Programmiersprachen:
Erlang / Elixir
& das Aktorenmodell

Fokusthemen dieses Abschnitts

- "Perfekte Kapselung" und klar definierte Datenabhängigkeiten durch …
  - Charakteristik der Sprache (funktionale Sprache)
  - Aktorenmodell (kein Shared Data, …)
- Deshalb: "Enorme Möglichkeiten" für Nebenläufigkeit.
  - Parallelisierung der "Kapseln".
- "Enorme Möglichkeiten" für "andere operative Themen":
  - Fehlerhandling: *fehlerhafte "Kapseln" ersetzen. Fehlerhandling-Strategie aus Library.*
  - Softwarehandling: Software Upgrade im laufenden Betrieb.
    "Kapseln" im laufenden Betrieb ersetzen.
  - Handling der "Kapseln" von ihrer internen Logik trennen.
    Handling aus Library (== OTP) beziehen.

Aktorenmodell

… Modell für nebenläufige Programme.

Nebenläufige Einheiten: Aktoren.

Verwendung unter anderem in Erlang, Elixir, Scala.
Oder z.B. in JVM-basierten Sprachen mittels des Akka Frameworks.

Aktoren & Nachrichten

Aktoren können drei verschiedene Reaktionen durchführen:

1. Nachrichten an sich selbst oder an andere Aktoren verschicken.
2. Neue Aktoren erzeugen.
3. Das eigene Verhalten ändern (ihren Code / Berechnungen ausführen).

Nachrichtenaustausch asynchron:
Sender kann sofort mit anderen Aktionen fortfahren.

KMPS Steckbrief:
Die Programmiersprache **Erlang**

… von Ericsson („**Er**icsson **Lang**uage“) für Programmierung
fehlertoleranter, verteilter, komplexer Systeme.

Komplett in Erlang programmierte Ericsson Systeme seit 1998.

Erlang Charakteristika

Funktionen als Aktoren. Preemptive scheduling.

Erlang VM:s als „Network System“. Funktioniert & skaliert gut.

Mnesia: Verteilte Datenbank.
OTP: Open Telecom Platform (OTP) Library.

Charakteristik passt optimal zu Server-Systemen.

Erlang & GPRS SGSN, GGSN, SGSN-MME

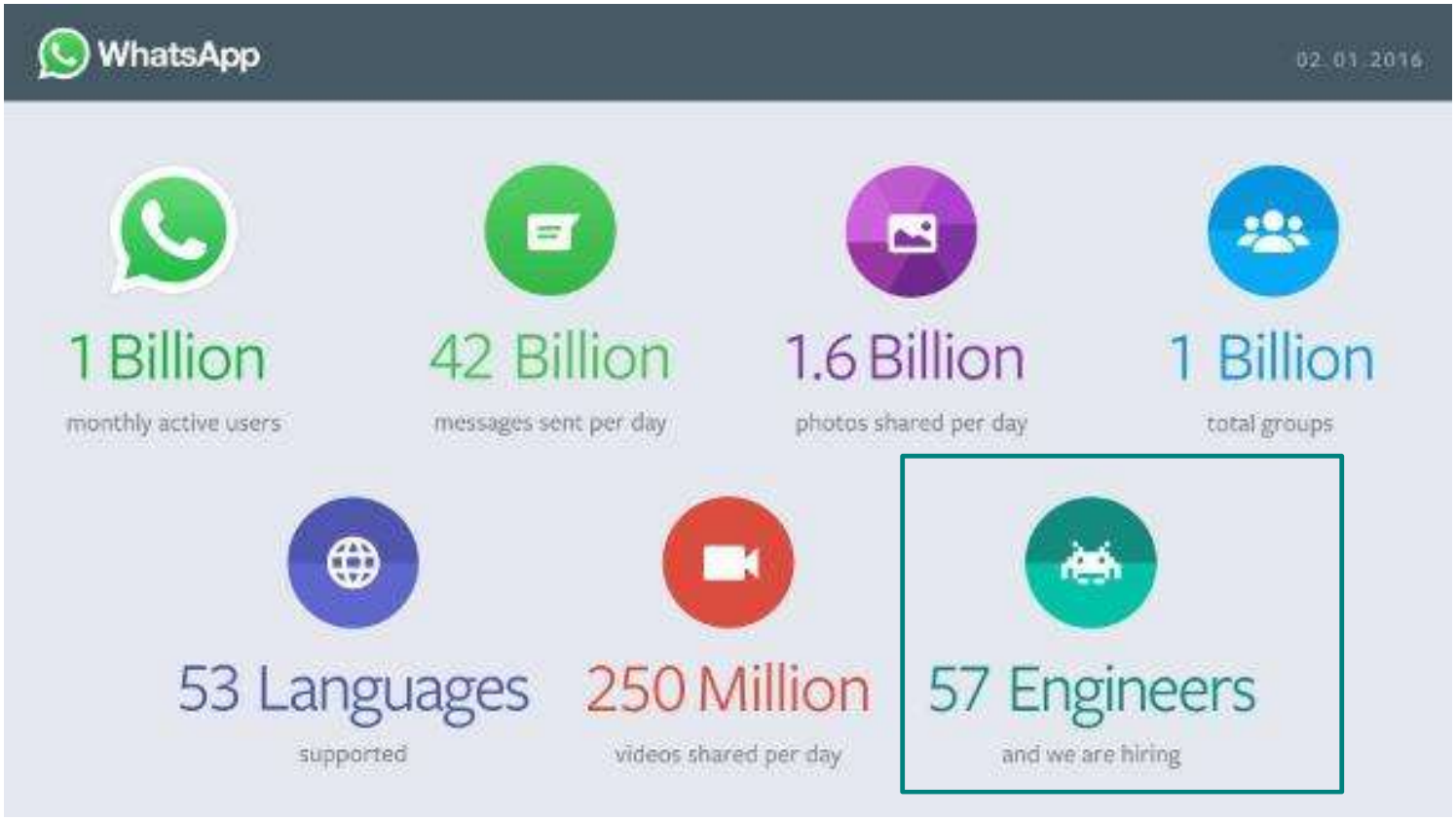
THE ERICSSON SGSN-MME
-
Over a Decade of Erlang Success

http://www.erlang-factory.com/upload/presentations/597/sgsn.pdf

Bildquelle: http://www.ericsson.com/ourportfolio/products/sgsn-mme

WhatsApp: Nur 57 IT Experten für solch ein System …

(Daten aus 2016)



Bildquelle: http://heise.de/-3089551 , 2016

"Scaling to the next level at WhatsApp" (2014)

A session at Erlang Factory SF Bay Area 2014
Rick Reed, Friday 7th March, 2014, 2:00pm to 2:50pm (PST)

… hundreds of nodes, thousands of cores, hundreds of terabytes of RAM, …
The Erlang/FreeBSD-based server infrastructure at WhatsApp.

(…)

>8000 cores, >70M Erlang messages per second:
Erlang … a versatile, reliable, high-performance platform.

Quelle: http://lanyrd.com/2014/erlangfactory/scwqrt/

GitHub helped Erlang and Erlang helped GitHub

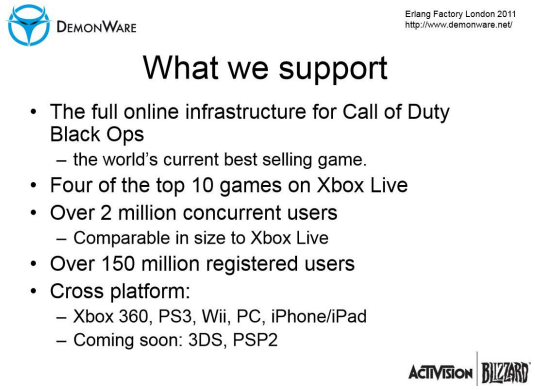
Cf. https://www.infoq.com/interviews/erlang-and-github
Tom Preston-Werner, GitHub CTO:

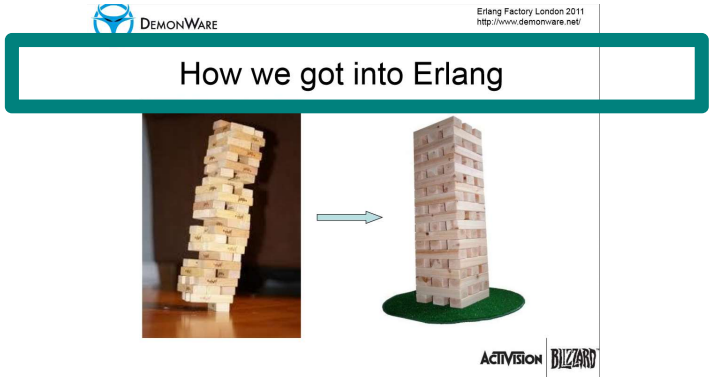
… Erlang to let frontend communicate to backend … be able to scale each of those layers separately …

… Erlang makes writing those kinds of servers just ridiculously simple.
I would never write a server like that in any other languages,
Erlang is so perfect for that …

DemonWare, Multiplayer Game Server

… für die *Call of Duty* Reihe





Quelle: http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf

Die Sprache Elixir

Relativ junge Programmiersprache. Basiert auf der Erlang VM.

http://elixir-lang.org/
https://github.com/elixir-lang/elixir

Atoms (Symbolische Konstanten) in Erlang

… sind eindeutige, konstante, symbolische Werte.

```
Temperatur = { 25.0, celsius }.
```

Scala: symbolic literals. `'hallo`

Nebenläufigkeit / Parallelismus in Erlang

- Leicht-gewichtige Erlang Prozesse.
- Entkoppelte Prozesse. Nachrichten. Keine (Shared Data) Seiteneffekte.
- Erlang VM, Verteilung von Erlang Code über viele parallele Rechner leicht möglich und im Programm nicht spezifisch sichtbar.

Ein einfaches Berechnungsmodul, in Erlang …

```
area({square, X})    ->   X * X;

area({rectangle, X, Y}) -> X * Y.
```

Patterns *Actions*

Die `area()` Funktion nimmt ein *Pattern* als Argument und berechnet abhängig davon das Resultat.

Umwandlung in einen Server, in Erlang …

```
area() ->
    receive
        {square, X} -> X * X;
        {rectangle, X, Y} -> X * Y
    end,
    area().
```

Nachrichtenempfang

Patterns => Nachrichteninhalt

Endlosschleife (tail recursion).

Resultat zurück zum Aufrufer schicken, in Erlang …

```
Pid = spawn(server2,
            area, []).

Pid ! {self(),
       {square, 10}}.

receive
   Reply -> Reply
end.
```

```
-module(server2).
-export([area/0]).

area() ->
   receive
      {From, {square, X}} ->
         From ! X * X;
      {From, {rectangle, X, Y}} ->
         From ! X * Y
    end,
    area().
```

Der Aufrufer / Client schickt seine Prozess-ID `self()` mit.
Der Server schickt das Resultat per Nachricht an diesen Client zurück, wo es in `Reply` gespeichert und als "Resultat des Clients" von diesem als sein Rückgabewert zurückgegeben wird.

Unterscheidung mehrerer Server, in Erlang …

```
Pid = spawn(server3,
            area, []).

Pid ! {self(),
       {square, 10}}.

receive
   {Pid, Reply} -> Reply
end.
```

```
-module(server3).
-export([area/0]).

area() ->
   receive
      {From, {square, X}} ->
         From ! {self(), X * X};
      {From, {rectangle, X, Y}} ->
         From ! {self(), X * Y}
   end,
   area().
```

Der Server schickt seine Prozess-ID `self()` mit.
Der Client macht Pattern Matching auf empfangene Nachrichten und matcht nur Nachrichten, die von der ID des selbst kontaktierten Servers stammen.
Das bisherige `receive` matchte auf jede Nachricht!
Jetzt kann Client die Antworten verschiedener Server seinen Anfragen zuordnen.

Viele Anfragen eines Clients an einen Server, in Erlang …

```
Pid = spawn(server4,
            area, []),
Tag = erlang:make_ref(),
Pid ! {self(), Tag,
       {square, 10}},
receive
   {Pid, Tag, Reply} ->
      Reply
end.
```

```
-module(server4).
-export([area/0]).

area() ->
   receive
      {From, Tag, {square, X}} ->
         From ! {self(), Tag, X * X};
      {From, Tag, {rectangle, X, Y}} ->
         From ! {self(), Tag, X * Y}
    end,
    area().
```

Client generiert eindeutige ID für die Anfrage, sendet diese mit.
Der Server sendet diese ID in der Response mit zurück.

Bei mehreren Requests *gegenüber dem gleichen Server* dadurch klar, zu welcher Anfrage diese Response gehört.

Registrierung der Server in der "Erlang Global Registry", in Erlang …

```
% Einmalig:
Pid = spawn(server5, area, []).

global:register_name(server5, Pid).


% Ggfs wiederholt:

Pid = global:whereis_name(server5).

Tag = erlang:make_ref().
Pid ! {self(), Tag, {square, 10}}.
receive
    {Pid, Tag, Reply} -> Reply
end.
```

```
-module(server5). % identisch zu server4
-export([area/0]).

area() ->
   receive
      {From, Tag, {square, X}} ->
         From ! {self(), Tag, X * X};
      {From, Tag, {rectangle, X, Y}} ->
         From ! {self(), Tag, X * Y}
    end,
    area().
```

Nur der Erzeuger eines Prozesses kennt diesen (dessen Prozess-ID) eigentlich. Durch die globale Registry können Clients Server-ID:s *(== "Service-IDs")* herausfinden. Registrierung unter Aliasnamen. Konvention: Benutze den Modulnamen als Alias.

Erlang Aktorenmodell: Ein "Universal Server" als weiteres Beispiel, in Erlang …

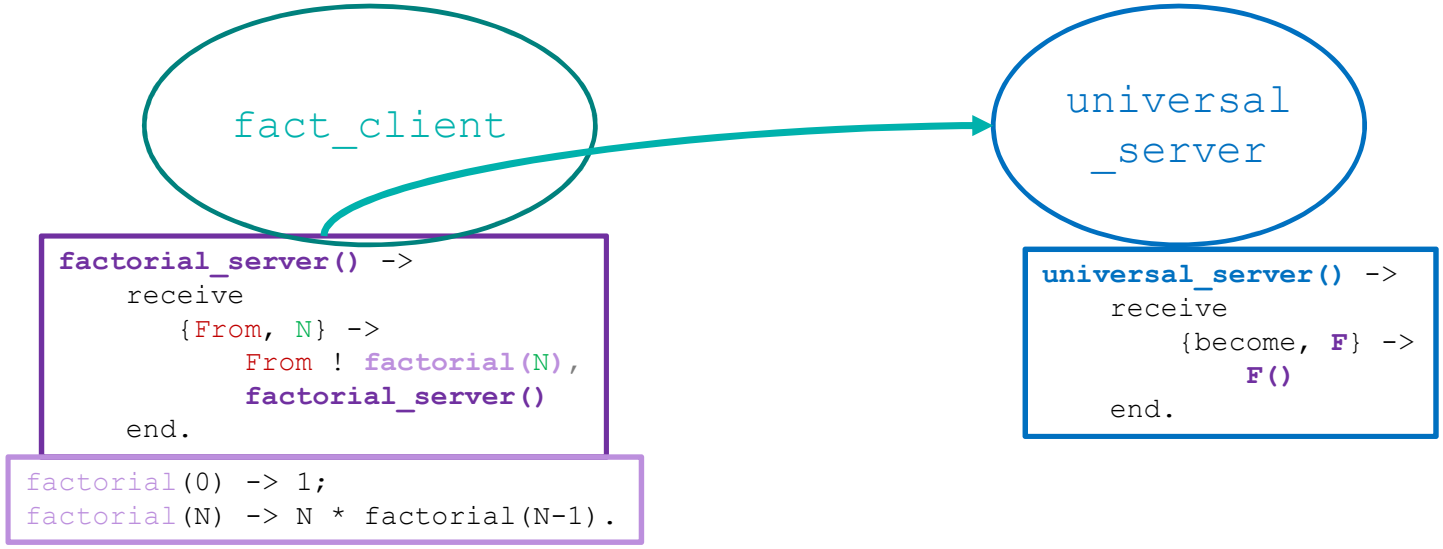
```
% File: universal_server.erl

-module(universal_server).
-export([universal_server/0]).


universal_server() ->
    receive
        {become, F} ->
            F()
    end.
```

Annahme:
`F()` beinhaltet die komplette "Server-Funktionalität":
Empfang von Berechnungs-Requests, "Endlosschleife" des Wartens auf Requests, …

Siehe auch: http://joearms.github.io/2013/11/21/My-favorite-erlang-program.html

Erlang Aktorenmodell: Ein "Universal Server" als weiteres Beispiel, in Erlang …

Dieser „Universal Server" wird jetzt durch einen Client zu einem Server für Fakultätswerte gemacht…



Vorteile des "viele kleine Prozesse" Ansatzes von Erlang

… d.h. mittels Aktorenmodell & light-weight processes:

- Skaliert gut.
- Sehr fehlertolerant: Viele kleine Einzelprozesse, bei Softwarefehler crasht nur einer dieser Prozesse, nicht das Gesamte.

"We do not have ONE web-server handling 2 millions sessions.
We have 2 million webservers handling one session each."

Quelle: http://joearms.github.io/2016/03/13/Managing-two-million-webservers.html

Für hohe Verfügbarkeit: Prozesse überwachen sich gegenseitig

Erlang Philosophie der Fehlerbehandlung: ***"Let it Crash"***

Fehlerbehandlung bei Parallelen Prozessen: Process Links

… verknüpfen Prozesse, die gemeinsam eine Aufgabe lösen sollen.

Links werden mittels der `link()` Funktion erzeugt.

Crashed einer der Prozesse, so werden die linked Prozesse auch beendet.
Bidirektional, d.h. in beide Richtungen.

"… rather like … a transaction: either … do what … supposed to do or … all killed."
Joe Armstrong, "Programming Erlang"

Begrenzung der Fehlerpropagation

… durch **System Processes** als "Crash-Firewall Prozesse".
Werden über Fehler benachrichtigt, sterben aber nicht.

Mittels `process_flag(trap_exit, true).` wird Prozess zum *System Process*.

Bekommt im Fehlerfall eine Nachricht `{ 'EXIT', Pid, Reason }`

… wobei …

`Pid`: PID des gestorbenen linked processes,
`Reason`: Fehlerursache

Empfang und Verarbeitung der EXIT Nachricht

```
-module(client2_link2).
-export([start/0, do_it/1]).
start() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(server4, area, []),
    register(server4, Pid).
do_it(X) ->
    server4 ! {self(), {square, X}},
    receive
        {'EXIT', _Pid, Reason} -> {error, Reason};
        Reply      -> Reply
        after 1000 -> timeout
    end.
```

aus vorherigem Beispiel

ist keine Zahl, kann also nicht mit sich selbst multipliziert werden …

```
$ erl
1> self().
<0.33.0>
2>
false
3> client2_link2:start().
true
4> client2_link2:do_it(ccc).
{error,{badarith,[{server2,area,0,
                           [{file,"server2.erl"},
                            {line,9}]}]}}

5>
=ERROR REPORT==== … ===
Error in process <0.36.0> with exit value:
{badarith,[{server2,area,0,[{file,"server2.erl"},{line,9}]}]}
6> self().
<0.33.0>
```

Der gecrashte server2 Prozess wird trotzdem vom System reportet.
Aber der System Process könnte unbeeinträchtigt weiterarbeiten.

OTP (**Open Telecom Platform**) als Erlang Framework für (Server) Applikationen

… realisiert Prozesshandling, Software Upgrade etc.

Ursprung in Telekommunikation, aber für jede Art von Server Software mit parallelen Erlang Prozessen verwendbar.

Code Struktur für Prozesse bei Nutzung von OTP

Für Modularität und Wiederverwendbarkeit:
Code der Prozesse in zwei Teile (Module) aufgeteilt.

- Generischer Teil: Generelles Verhalten, wiederverwendbar.
  Für generische Teile des Error Handlings, des Software Upgrade Handlings etc. …

- Spezifischer Teil mit der Applikations-Funktionalität:
  Das **Callback Module**.

OTP **Supervisor** Prozesse & **Worker** Prozesse

Aufbauend auf (normale / "Non-System") Processes und System Processes …
Worker technisch realisiert mittels (normalen / "non-system") processes.
Supervisors realisiert mittels system processes.
Beide linked miteinander.

Supervisors starten, stoppen und monitoren ihre Kind-Prozesse.

Ziel: Supervisor startet Kind-Prozesse bei Bedarf neu. Hält sie "lebend & gesund".

Worker für normale Berechnungen, ohne aufwändige Fehlerbehandlung.

OTP **Supervision Trees** & **Supervision Strategy**

Supervisors überwachen Worker und/oder andere Supervisors.

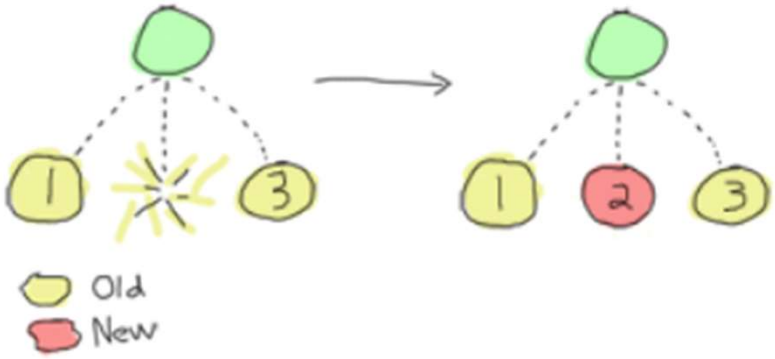
Ergibt baumartige Struktur.

Supervision Strategy:
Welche Restart-Strategie an bestimmter Stelle im Baum, wenn Kind-Prozess stirbt?

Restart Strategien: `one_for_one`

… if one of the workers fails, only that one will be restarted by the supervisor.

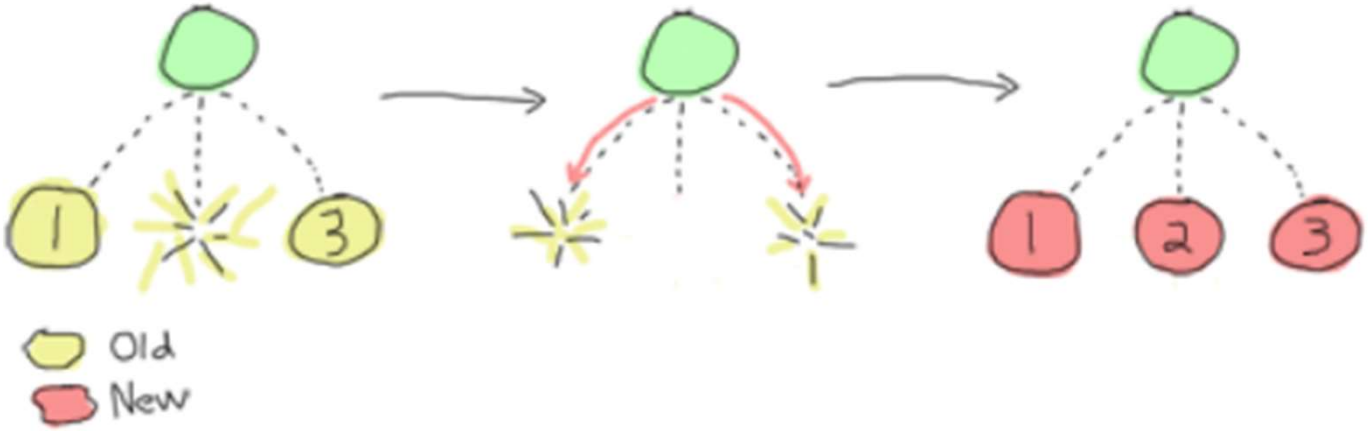
For independent processes.



Quelle: http://learnyousomeerlang.com/supervisors

Restart Strategien: `one_for_all`

… when all processes under a single supervisor heavily depend on each other.

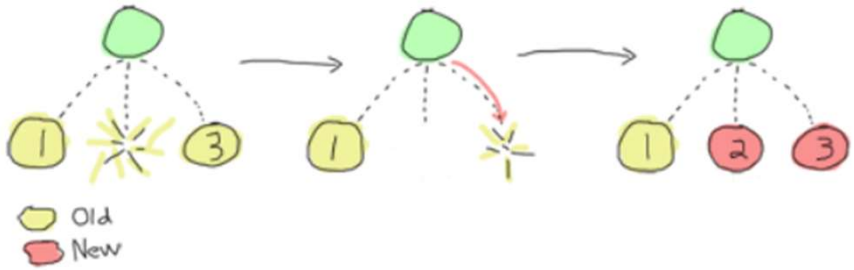


Quelle: http://learnyousomeerlang.com/supervisors

Restart Strategien: `rest_for_one`

… for processes that depend on each other in a chain.

If a process dies, all the ones that were started after it (depend on it) get restarted, but not the other way around.



Quelle: http://learnyousomeerlang.com/supervisors

Software Upgrade oder Software Patching in OTP, im laufenden Betrieb der Software

… wird durch OTP auch in industriellen Anwendungen möglich gemacht.

Beispiele aus der industriellen Praxis:
Live-Patching von Telekommunikations-Knoten,
des WhatsApp Servers,…

Beispielfunktion für den Software Upgrade

… sei eine Applikations-Funktion `my_func_call()`.

Version V0:
Keine Parameter, ganze Zahl als interne Daten (initial 0), "Wert um 1 hochzählen" als Logik, "aktueller Zustandswert + 5" als Rückgabewert.

Als Pseudo-Code (`gen_server` als Framework):

```
gen_server::init::my_func_call() -> { ok , internal_state = 0 }.

gen_server::call::my_func_call( internal_state = Num ) ->
    { reply, return_value = Num+5 , internal_state = Num+1 }.
```

Beispielfunktion für den Software Upgrade

Version V1:
Funktion hat jetzt einen numerischen Parameter.
Interner Zustand jetzt Tupel aus Literal und ganzer Zahl.
Migration der Datenwerte nötig!
Rückgabewert jetzt "aktueller Zustandswert + 10".
Logik jetzt "erhöhe Zustandswert um Parameterwert".

D.h. Parameter, Rückgabewert, interne Daten und Logik haben sich geändert …

Als Pseudo-Code:

```
gen_server::init::my_func() -> { ok , internal_state = {my_new_format,0} }.

gen_server::call::my_func_call( NewArg1 , internal_state = {my_new_format,Num} ) ->
    {reply , return_value = Num+10 , internal_state = {my_new_format,Num+NewArg1} }.

gen_server::code_change::my_func( old_version = "0", old_internal_state = Num, _Extra) ->
    {ok, {my_new_format, Num}}.
```

Software Upgrade in OTP: Beispiel

Version V0 des Moduls (Ausschnitt)

```
%% coding: latin-1
-module(mymodule).
-version("0").

-export([start_link/0, my_func/0]).

% Funktionsaufrufe "umleiten" über den gen_server:call Mechanismus.
% Ermöglicht paralleles Handling von neuen und alten Versionen von Funktionen ...
% Nutzer des Moduls können einen normalen Funktionsaufruf von my_func() durchführen ...
my_func() -> gen_server:call(?MODULE, my_func_call).

% Datenzustand (State) bei Initialisierung ...
init([]) -> {ok, 0}.

% Call: Synchrone Aufrufe ...
% Neuer Pattern Matching Fall des handle_call() aus gen_server ...
handle_call(my_func_call, _From, Num) -> {reply, Num+5, Num+1};

% Es gibt keine Vorgängerversion der V0, deshalb keine Code Change Implementierung hier ...
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

Codeausschnitt der relevanten Teile

Alte und neue state data des Moduls

Rückgabewert des Funktionsaufrufs

Software Upgrade in OTP: Beispiel

Version V1 des Moduls (Ausschnitt)

```
%% coding: latin-1
-module(mymodule).
-version("1").

-export([start_link/0, my_func/1]).

% In V1 hat diese Funktion plötzlich ein Argument, in V0 hatte sie das nicht! ...
my_func(NewArg1) -> gen_server:call(?MODULE, {my_func_call, NewArg1}).

% Call: Synchrone Aufrufe ...
% Neues Format des State (Laufzeit-Datums) in V1: Tupel statt Zahl! ...
% Außerdem hat der Request jetzt plötzlich einen Parameter ...
handle_call({my_func_call, NewArg1}, _From, {my_new_format, Num}) ->
                                        {reply, Num+10, {my_new_format, Num+NewArg1}};

% Migration von V0 nach hier (V1), insbes Migration des Status-Datums:
% Num => {my_new_format, Num} ...
code_change("0", Num, _Extra) -> {ok, {my_new_format, Num}}.
```

Codeausschnitt der relevanten Teile

Alte und neue state data des Moduls (neues Format)

Funktion hat jetzt einen Parameter!

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

Software Upgrade in OTP: Beispiel

Log des Upgrades

```
$ erl
1> file:copy('mymodule_V0.erl', 'mymodule.erl').
{ok,936}

2> c(mymodule).
{ok,mymodule}

3> mymodule:start_link().
{ok,<0.41.0>}

4> % Aufruf der Funktion im alten Modul, im alten Format (ohne Argumente) ...
4> mymodule:my_func().
*DBG* mymodule got call my_func_call from <0.33.0>
*DBG* mymodule sent 5 to <0.33.0>, new state 1
5
5> mymodule:my_func().
*DBG* mymodule got call my_func_call from <0.33.0>
*DBG* mymodule sent 6 to <0.33.0>, new state 2
6
```

Software Upgrade in OTP: Beispiel

Log des Upgrades

```
6> % Neue Version des Moduls einkopieren ...
6> file:copy('mymodule_V1.erl', 'mymodule.erl').
{ok,1406}

7> compile:file(mymodule).
{ok,mymodule}

8> % Ausführung anhalten (Laufzeit-Daten-Zustand bleibt aber bestehen und wird später migriert!) ...
8> sys:suspend(mymodule).
ok

9> % Jetzt den "old code" Bereich freiräumen, so dass der dann freie
9> % Bereich genutzt werden kann, um die Version V0 dort zu bewahren,
9> % solange Code aus V0 sich noch in der Ausführung befindet ...
9> % Code aus der Version vor V0, der immer noch in Ausführung wäre,
9> % würde jetzt gekilled...
9> code:purge(mymodule).
false
```

Software Upgrade in OTP: Beispiel

Log des Upgrades

```
10> code:load_file(mymodule).
{module,mymodule}

11> % Initiiere den Code Change. Ruft code_change/3 in der (neuen) V1 des
11> % Moduls auf. Diese Funktion kann dann die Daten migrieren, während
11> % das Processing suspendiert ist ...
11> sys:change_code(mymodule,mymodule,"0",[]).
ok
12> sys:resume(mymodule).
ok

13> mymodule:my_func(10).
*DBG* mymodule got call {my_func_call,10} from <0.33.0>
*DBG* mymodule sent 12 to <0.33.0>, new state {my_new_format,12}
12
14> mymodule:my_func(100).
*DBG* mymodule got call {my_func_call,100} from <0.33.0>
*DBG* mymodule sent 22 to <0.33.0>, new state {my_new_format,112}
22
```