Introduction to Resilient Distributed Programming in Erlang

LINFO1104 and LINGI1131
May 12 and May 13, 2020

Peter Van Roy
Université catholique de Louvain

This lecture is based on information taken from many Erlang documents
Prerequisites for the lecture are some knowledge of functional programming and message passing

1

Overview

- Erlang performance
  - Some numbers to show Erlang's strengths
- Basic concepts of the Erlang language
  - Pure functional core with dynamic typing
  - Process (agents) and message passing with mailboxes
  - Failure detection with linking and monitoring
  - Dynamic code updating
- Programming abstractions of the Erlang/OTP platform
  - Basic principles of resilient system design
    - Erlang stable storage: ETS, DETS, Mnesia
    - Testing tools: EUnit, Common Test, Dialyzer
  - Standard behaviors (concurrency patterns)
    - Generic servers, generic FSMs, generic event handling, supervisor trees
    - Principles of a generic server
    - Principles of supervisor trees
- Conclusions

2

Erlang introduction

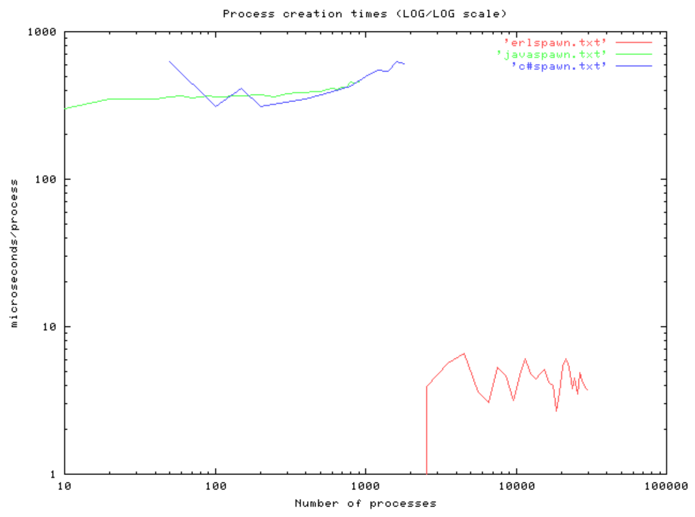
- Erlang was developed in Ericsson for telecommunications in 1986 (Java is from 1991)
  - It is released as OTP (Open Telecom Platform) with a full set of libraries
  - It is supported by Ericsson, the Erlang Ecosystem Foundation, and a user community (www.erlang.org)
- Erlang programs consist of lightweight "processes", which are active agents that communicate using asynchronous FIFO message passing ("actor model" as proposed by Carl Hewitt in 1973)
  - Erlang processes share nothing: all data is copied between them
  - Erlang processes receive messages through a mailbox that is accessed by pattern matching. Messages can be received out of order if they match.
- Erlang/OTP supports reliable long-lived distributed systems using behaviors and supervisors
  - Behaviors are generic concurrency patterns that make it easy to write robust concurrent systems
  - Supervisors are a general pattern to observe processes and restart them when they crash
  - Ericsson AXD 301 ATM switch with 1.7 million lines of Erlang claims 99.9999999% availability (one may doubt the number of 9's, but the system is extremely available!)

3

Erlang performance

4

Erlang process creation times

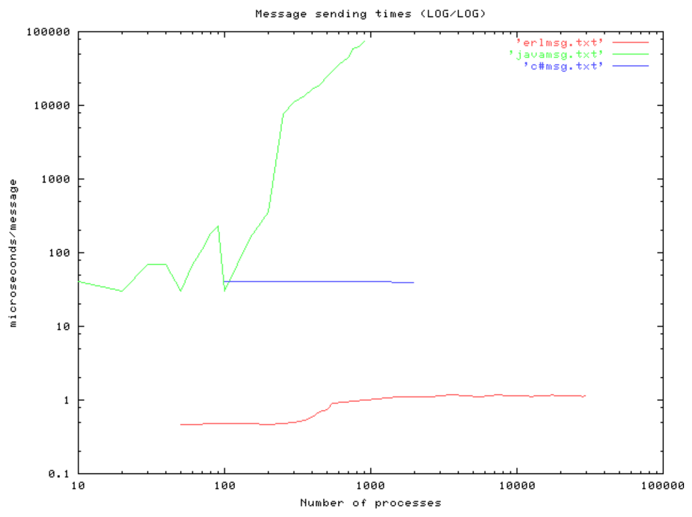


From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- We compare process creation times for Erlang, Java, and C#
- These numbers were measured in 2002

5

Erlang message sending times

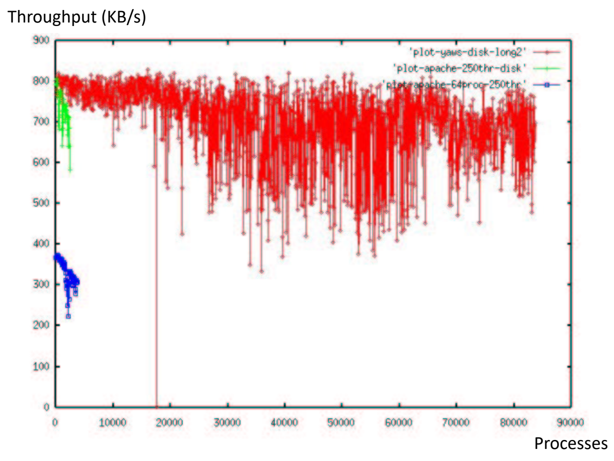


From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- We compare message sending times for Erlang, Java, and C#
- These numbers were measured in 2002

6

Use case: web server

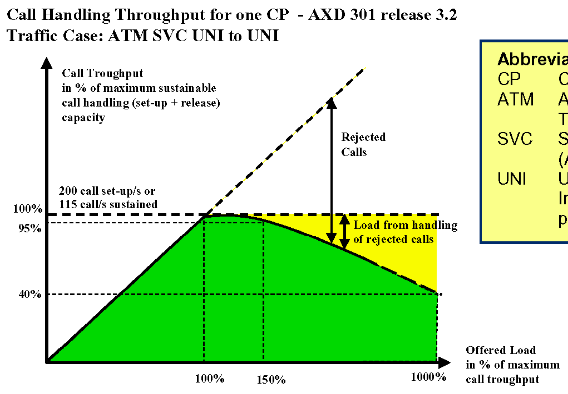


From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- Throughput versus number of processes for Web servers
  - Red = yaws (Yet Another Web Server, in Erlang on NFS)
  - Green = apache (local disk)
  - Blue = apache (NFS)
- Yaws: 800 KB/s up to 80,000 processes
- Apache: crashes at around 4,000 processes

7

Use case: AXD301 Erlang-based ATM switch



Abbreviations:

CP	Control Processor
ATM	Asynchronous Transfer Mode
SVC	Switched Virtual (ATM) Channel
UNI	User-Network Interface signaling protocol

From Ulf Wiger, Four-fold Increase in Productivity and Quality, 2001.

- The AXD 301 is a general-purpose high-performance ATM switch from Ericsson
  - The AXD 301 is built using Erlang OTP supplemented with C and Java
- Throughput drops linearly when overloaded
  - 95% throughput at 150% load, descending to 40% throughput at 1000% sustained load
- AXD 301 release 3.2 has 1MLOC Erlang, 900KLOC C/C++, 13KLOC Java
  - In addition, Erlang/OTP at that time had 240KLOC Erlang, 460KLOC C/C++, 15KLOC Java

8

Basic concepts

9

Pure functional core

- Within a process, Erlang runs as a pure functional language
  - All variables are single assignment (bound when they are declared)
  - Functions are values with lexically scoped higher-order programming
  - Pattern matching can be used in **case** and **if** statements (and **receive**)
- All data structures are symbolic values
  - Integers (arbitrary precision), floats, atoms (symbolic constants)
  - Lists [george,paul,john,ringo] and tuples {Key,Val,L,R}
  - Strings are lists of ASCII codes (integers)
  - Binary vectors (used for protocol computations)

10

Dynamic typing versus static typing

- Erlang is a strongly typed language, that is, types are enforced by the language
  - Many popular languages are strongly typed, such as Java, Scheme, Haskell, and Prolog
  - Weakly typed languages, e.g., C and C++, allow access to a type's internal representation
- Strongly typed languages can be dynamically or statically typed
  - Erlang is a dynamically typed language because variables can be bound to entities of any type
  - In a statically typed language, variable types are known at compile time
- Static typing allows catching more program errors at compile time
  - However, this does not mean that statically typed programs are more resilient
    - Static typing allows catching many "surface errors" but does not help with "deep errors"
  - Well-written Erlang programs are among the most resilient software artefacts ever built, because Erlang provides adequate mechanisms to overcome deep errors

11

An Erlang module

- The source code of an Erlang program is organized in modules:

```
-module(math).
-export([areas/1]).
-import(lists, [map/2]).

areas(L) -> lists:sum(map(fun(I) -> area(I) end, L)).

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

- Modules import and export, giving a dependency graph of modules

12

Creating processes and sending messages

- Any process can create another by calling spawn
  - Pid = spawn(Fun) : function Fun defines the behavior, Pid is the process name
  - Fun may be anonymous or named
    - `fun(args) -> expr end`
    - `fun name/arity`
- The process name Pid is a unique constant that identifies the process
- Messages can be sent to the process using the process name
  - Pid ! Message
  - Messages are sent asynchronously and all data in messages is copied
  - Messages can be received by the **receive** statement

13

Receiving messages

- Each process has a mailbox that contains an ordered list of messages received by the process
- Messages are extracted from the mailbox using the **receive** statement
  - The **receive** uses pattern matching to remove the first message that matches
  -
    ```
    receive
      pattern1 when guard1 -> expr1;
      pattern2 when guard2 -> expr2;
      …
      patternN when guardN -> exprN
    end
    ```

14

Send and receive

```
Pid ! Message,
…

receive
     Message1 ->
      Actions1;
     Message2 ->
      Actions2;
     …
     after Time ->
      TimeOutActions
end
```

15

Receive mailbox semantics

- When a process executes **receive**:
  - If the mailbox is empty, the **receive** blocks and waits for a message
  - If the mailbox is not empty, it takes the first message and tries patterns in order starting from the first, if it finds a matching pattern it executes the corresponding code
  - If no pattern matches, the **receive** blocks and waits for the next message
    - Unmatched messages remain in the mailbox and can be removed by future **receive** calls
    - This allows different parts of a process to treat different kinds of messages
    - Messages can be removed out-of-order (in a different order from when they arrived)
    - Care must be taken that messages do not stay in the mailbox forever (memory leak)
- Patterns are symbolic data structures containing variable identifiers and guards are simple built-in tests

16

Process registering

- A process Pid can be registered with a name, which is an atom, to make the process globally available
- register(Atom, Pid) : give Pid the global name Atom
- unregister(Atom) : remove the registration for Atom
- whereis(Atom) -> Pid | undefined : returns the Pid of a registered process, or undefined if no such process exists
- registered() -> [Atom :: atom()] : returns a list of all registered processes

17

Process linking

- Two processes can be linked together
  - Process Pid1 calls link(Pid2) or conversely; linking is bidirectional
- Process termination: send exit signal
  - A process terminates with an exit reason, which is sent as a signal to all linked processes
  - When a process terminates normally, the exit reason is the atom normal, otherwise when there is a run-time error, the exit reason is {Reason,Stack}
- Propagating process termination: transitive by default (“link set”)
  - The default behavior when a process receives an exit signal with reason other than normal is to terminate and to send exit signals with the same reason to its linked processes
  - A process can be set to trap exit signals by calling process_flag(trap_exit,true)
  - A received exit signal is then transformed into the message {‘EXIT’, FromPid, Reason}, which is put into the mailbox of the process

18

Using process linking

- If a process throws an exception that is not caught at the top level, then the process terminates and broadcasts its exit signal to all linked processes
- A few additional operations are defined to manage this:
  - exit(Pid,Why): send an exit signal to Pid without terminating
  - exit(Pid,kill): send an unstoppable exit to process Pid

```
start() -> spawn(fun go/0).

go() ->
      process_flag(trap_exit, true),
      loop().

loop() ->
      receive
        {'EXIT',Pid,Why} -> …
      end.
```

19

Process monitoring

- Process monitor is an asymmetric version of linking
  - No resemblance to monitor concept in shared-state concurrency!
- For example, in a client/server, if the server crashes we want to kill the clients, but if a client crashes we do not want to kill the server
- If process Pid1 executes:

  Ref = erlang:monitor(process, Pid2)

- Then if Pid2 dies with exit reason Why, then Pid1 will be sent a message {'DOWN',Ref,process,B,Why}

20

Distributed Erlang

- A distributed Erlang system consists of a number of Erlang runtime systems, called *nodes*, communicating with each other
- Message passing between processes at different nodes, as well as links and monitors, is transparent when using Pids
- Nodes can spawn processes on other nodes using spawn(Node,M,F,A)
- Registered names are local to each node: both node and name must be specified when sending messages using registered names
- The first time the name of a node is used, a connection attempt is made to the node; connections are made transitively giving a fully connected mesh by default (recent versions of Erlang allow more scalable connection topologies)

21

Dynamic code change (1)

- In a real-time system, we would like to change the code without stopping the system
  - Some systems are never supposed to be stopped, e.g., the X2000 satellite control system developed by NASA
  - Hot code changing is difficult in a monolithic programming system, however, Erlang makes it possible because processes are independent (no sharing)
- Erlang allows for each module to have two versions of code
  - All new processes will be dynamically linked to the latest version
  - If the code is changed, then processes can choose to continue with the old code or to use the new code
  - The choice is determined by how the code is called

22

Dynamic code change (2)

- Call the new version (if available)

```
-module(m).

loop(Data, F) ->
     receive
     {From,Q} ->
          {Reply,Data1}=F(Q,Data),
          m:loop(Data1, F)
     end.
```

Use new version

- Keep calling the old version:

```
-module(m).

loop(Data, F) ->
     receive
     {From,Q} ->
          {Reply,Data1} =F(Q,Data),
          loop(Data1, F)
     end.
```

Use old version

- This mechanism is used by libraries that manage upgrading of application releases

23

Client/server process with hot code swap

- Dynamic code change can also be done for individual processes using higher-order functions

Process code

```
server(Fun, Data) ->
      receive
         {new_fun,Fun1} ->
             server(Fun1,Data);
         {rpc,From,ReplyAs,Q} ->
             {Reply,Data1} =
                   Fun{Q,Data},
             From!{ReplyAs,Reply},
             server(Fun, Data1)
      end.
```

```
rpc(A,B) ->
      Tag=new_ref(),
      A!{rpc,self(),Tag,B},
      receive
          {Tag,Val} -> Val
      end.
```

24

Programming abstractions

25

Erlang philosophy

- How can we make robust software?
  - Popular languages (e.g., Java and Python) are inadequate
- Principles of robust software (from Joe Armstrong’s Ph.D. thesis)
  - Errors cannot be fully eliminated, therefore they must be handled (both hardware and software errors)
  - Software components are the units of failure: errors occurring in one will not affect others (“strong isolation”)
  - Software should be fail-fast: either function correctly or stop quickly
  - Failure should be detectable by remote components
  - Software components share no state, but communicate through messages

26

Erlang “slogans”

- “Let it crash”, “If you can’t do your job, crash”
  - Instead of trying to fix things when errors happen, which leads to a large number of complicated states, instead map everything to one simple state, namely “crashed”
- “Let some other process do error recovery”
  - Both hardware and software errors can occur and trying to solve them in the process makes things complicated, better to detect and handle any error, hardware or software, elsewhere
- “Do not program defensively”
  - Defensive programming means to add checks as much as possible. This is not productive since it makes the program complicated (in particular, what do you do when a check fails?) and it will not remove all errors. Errors will still occur and still need to be handled. The best way is to map all errors no matter how bizarre to a single fault state, namely “crashed”.

27

Erlang/OTP systems

- Erlang/OTP supports a hierarchy of systems:
  - Release: Contains all the information necessary to build and run a system, including a software archive and a set of procedures for installation (including upgrading without stopping)
  - Application: Contains all the software necessary to run a single application, not the entire system. Releases are often composed of multiple applications that are largely independent of one another, or that are hierarchically dependent.
  - Behavior: A set of processes that together implement a concurrency pattern
    - A notable behavior is supervisor: a tree of processes whose purpose is to monitor behaviors and each other and restart them when necessary
  - Worker: A process that is an instance of a behavior, usually instances of gen_server, gen_event or gen_fsm

28

Using behaviors to abstract concurrency and fault tolerance

- In general, program code can be structured into “difficult” and “easy” modules
  - The difficult modules should be few and written by expert programmers
  - The easy modules should be many and written by regular application programmers
- Concurrency and fault tolerance are difficult to implement correctly
  - Behaviors: generic components provided by Erlang to hide concurrency and fault tolerance
  - Behaviors are Erlang’s “programming patterns” for building robust concurrent programs
- The Erlang/OTP platform provides library support for many important behaviors
  - See Erlang/OTP System Documentation, Ericsson, Version 10.7, March 15, 2020

29

Standard Erlang/OTP behaviors

- The Erlang/OTP platform provides the following standard behaviors:
  - Generic server (gen_server): to build client/server architectures with registration, start/stop, timeouts, state management, synchronous/asynchronous calls, error handling
  - Generic event handler/manager (get_event): event handlers, such as loggers, to respond to a stream of events, handling them and sending notifications
  - Generic finite state machine (gen_fsm): applications (e.g., protocol stacks) can be modeled as finite state machines, which provides a set of rules State × Event → Actions × State
  - Application: a component that can be started and stopped as a unit, and can be reused in other systems
  - Supervisor: the generic toolkit for implementing supervisor hierarchies
- These behaviors hide most of the complexity of each concept, in particularly the concurrency and fault tolerance are vastly simplified using behaviors
  - All non-supervisor behaviors are designed to be pluggable into a supervisor hierarchy

30

Erlang stable storage

- Resilient systems need stable storage to survive crashes
  - When a process is restarted by a supervisor, it uses the stable storage to start in a consistent state
  - Stable storage and supervisor trees are the two pillars of Erlang's resilience
- Erlang provides three levels of stable storage
  - ETS (Erlang Term Storage): Efficient in-memory storage for arbitrary Erlang terms, with limited size and concurrency abilities
  - DETS (Disk-based ETS): Same abilities as ETS, but stored on disk
    - ETS and DETS are limited to single nodes (single Erlang virtual machine)
  - Mnesia: A transactional database that is built on top of ETS and DETS and allows making a balance between ETS efficiency and DETS persistence
    - Mnesia supports distribution and replication on multiple nodes

31

Testing

- It's not enough to design for resilience, testing is essential
- Erlang provides a spectrum of practical and powerful testing tools
  - EUnit: Unit testing framework, including test generators (using higher-order programming to generate new tests) and fixtures (scaffolding around tests)
    - Supports test-driven development (TDD)
    - Good for testing modules and libraries
  - Common Test: Full-featured system testing framework
    - Test groups: allows running tests in parallel or in random order, for race conditions
    - Test suites: for handling dependencies between applications when testing
    - Test specifications including simulating of abnormal termination (fault injection)
  - Dialyzer: Dynamic type checker based on success types
    - Will not make a proof of correctness, but any type error it finds is a real error

32

Generic servers

33

Generic server example

- Client/server code can be written as a generic part plus a specific part
- Consider the following simple (non-generic) server, which keeps track of "channels" that can be allocated and deallocated

```
-module(ch1).
-export([start/0]).
-export([alloc/0,free/1]).
-export([init/0]).

start() -> spawn(ch1, init, []).

alloc() -> ch1 ! {self(),alloc},
     receive {ch1, Res} -> Res end.

free(Ch) -> ch1 ! {free, Ch}, ok.
```

```
init() ->
     register(ch1, self()),
     Chs=channels(),
     loop(Chs).

loop(Chs) ->
     receive
        {From,alloc} ->
            {Ch, Chs2} = alloc(Chs),
             From!{ch1,Ch},
             loop(Chs2);
        {free,Ch} ->
             Chs2=free(Ch, Chs),
             loop(Chs2)
     end.
```

34

Generic server: generic part

- The server code can be rewritten as a generic part plus a callback module
- We give first the generic part

```
-module(server).
-export([start/1]).
-export([call/2,cast/2]).
-export([init/1]).

start(Mod) ->
    spawn(server,init,[Mod]).
call(Name,Req) ->
    Name ! {call, self(), Req},
    receive {Name,Res} -> Res end.
cast(Name,Req) ->
    Name ! {cast, Req}.
```

```
Init(Mod) ->
     register(Mod, self()),
     State=Mod:init(),
     loop(Mod, State).

loop(Mod, State) ->
     receive
       {call,From,Req} ->
          {Res,State2}=
            Mod:handle_call(Req,State),
          From ! {Mod, Res},
          loop(Mod, State2);
       {cast, Req} ->
          State2 =
            Mod:handle_cast(Req,State),
          loop(Mod, State2)
     end.
```

35

Generic server: callback module

- Here is the callback module
  - This is the only thing the programmer needs to write
  - Server name ch2 and protocol messages are hidden from clients

```
-module(ch2).
-export([start/0]).
-export([alloc/0,free/1]).
-export([init/0,
         handle_call/2,
         handle_cast/2]).
```

```
start() -> server:start(ch2).
alloc() ->
    server:call(ch2, alloc).
free(Ch) ->
    server:cast(ch2,{free,Ch}).
```

```
init() -> channels().
handle_call(alloc, Chs) ->
     alloc(Chs).
handle_cast({free, Ch}, Chs) ->
     free(Ch, Chs).
```

Connection between generic part (init, handle_call, handle_cast) and specific part (channels, alloc, free)

36

Channels implementation

- Here are channels, alloc, and free as used by the gen_server example
  - Channels manage a 2-tuple {A,F} where A is allocated channels and F is free channels

```
channels() -> {_Allocated=[], _Free=lists:seq(1,100)}.

alloc({Allocated, [H|T]=_Free}) -> {H, {[H|Allocated], T}}.

free(Ch, {Alloc, Free} = Channels) ->
     case lists:member(Ch, Alloc) of
        true ->
           {lists:delete(Ch, Alloc), [Ch|Free]};
        false ->
           Channels
     end.
```

37

Generic server behavior

- The Erlang/OTP gen_server extends the behavior of this example
- Starting a gen_server
  - The name of the callback module is specified
- Requests
  - Synchronous requests (call): there is a return value
  - Asynchronous requests (cast): no return value
- Stopping a gen_server
  - If the gen_server is part of a supervision tree, no stop function is needed. The server will automatically be terminated by its supervisor.
  - If the gen_server is standalone, a stop function may be useful

38

Supervisor trees

39

Supervisor trees introduction

- In practical concurrent and distributed systems, it is observed that most faults and errors are transient
  - For example: network problems, timing problems, concurrent startup
  - Simple retrying is a surprisingly successful strategy
- Supervisor trees are designed to favor this strategy
  - Process sets are “supervised” (observed for failure) by supervisor processes
  - Supervisors have authority to stop and restart supervised processes
  - Supervisors are themselves observed, in case they fail
- Supervisor trees are carefully implemented to avoid races
  - Starting of a supervisor tree is synchronous, to establish correct initial state

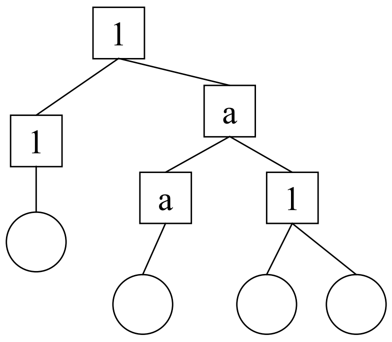
40

Supervisor structure and principles

- Supervisor tree is a hierarchy with a root
  - A supervisor tree consists of a set of supervisor nodes, organized as a hierarchy with a root node and internal nodes (the root is a very important process!)
  - A supervisor node is responsible for starting, stopping, and monitoring its child processes
  - All Erlang behaviors are designed to work together with supervisors
- Restart principles
  - Restart strategy: one_for_one, one_for_all, rest_for_one
  - Restart frequency: the number of restarts is limited per time interval
    - If the limit is exceeded, the supervisor terminates and the next higher level supervisor takes some action
    - The intention is to prevent a situation where a process dies repeatedly for the same reason and is always restarted

41

Supervision hierarchies



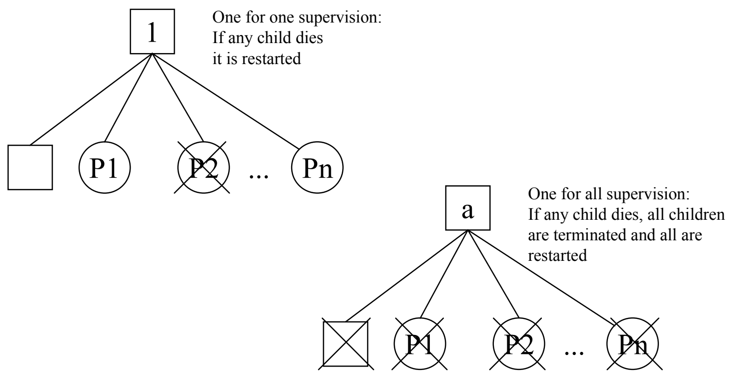
Abbreviations:

a	One-for-all supervision
1	One-for-one supervision

- A supervisor (☐) is a process whose sole purpose is to start, monitor, and possibly restart workers (O)
- A worker is an instance of a behavior, which raise exceptions when errors occur

42

One-for-one and one-for-all supervision



- Different forms of supervision depending on how the children processes work
  - One-for-one: if the children are independent (each manages one connection)
  - One-for-all: if the children are collaborating, then if one crashes they all have to be restarted, even the correct ones

43

Other behaviors

44

Generic finite state machine (gen_fsm)

- A finite state machine is defined as a set of relations of the form:
  State(S) × Event(E) → Actions(A) × State(S')
  - "If we are in state S and event E occurs, then we should perform the actions A and make a transition to state S' "
  - Many applications can be modeled as FSMs
- Starting a gen_fsm
- Notifying about events (for actions)
- Stopping a gen_fsm
  - If the gen_fsm is part of a supervision tree, no stop function is needed.
  - If the gen_fsm is standalone, a stop function may be useful

45

Generic event handler (gen_event)

- An event is a message that is sent when a specific condition occurs
  - For example, an error, an alarm, or information to be logged
- The event manager installs zero or more event handlers
  - When the event manager is notified about an event, all the event handlers will process it
- Starting and stopping an event manager
  - As before, if the gen_event is part of a supervision tree, no stop function is needed

46

Conclusions

47

Conclusions

- The Erlang/OTP platform combines the Erlang language with generic OTP libraries for building resilient highly concurrent and distributed systems (www.erlang.org)
  - Erlang supports “processes” (active agents) that share no state and communicate through asynchronous messages
    - Processes are defined using pure functional programming and support dynamic code updating
    - Failure detection is part of the message communication (process linking)
  - OTP supports resilient releases and applications using behaviors, supervisor trees, and testing
    - A behavior is a generic concurrency pattern (server, FSM, event handling)
    - A supervisor tree manages how failure handling is done
    - Testing tools for testing concurrent and distributed applications including fault injection
- Erlang/OTP is being used successfully for many industrial applications, and Erlang ideas are being incorporated into other programming languages and systems
  - Mainstream languages are extended to message passing and agents, and Erlang itself is evolving to resemble mainstream languages (Elixir provides a Java-like syntax with an Erlang semantics)
  - The Erlang Ecosystem Foundation was founded recently to support Erlang and Elixir

48

Bibliography

49

Bibliography

- Joe Armstrong, Concurrency Oriented Programming in Erlang, Talk slides, Nov. 2002.
- Joe Armstrong, Making Reliable Distributed Systems in the Presence of Software Errors, Ph.D. dissertation, KTH, Dec. 2003.
- Joe Armstrong, Programming Erlang: Software for a Concurrent World, The Pragmatic Bookshelf, 2007.
- Staffan Blau and Jan Rooth. AXD 301–A New Generation ATM Switching System, Ericsson Review No. 1, 1998.
- Francesco Cesarini and Steve Vinoski. Designing for Scalability with Erlang/OTP, O'Reilly, 2016.
- Ericsson AB. Erlang (Condensed) , Talk slides.
- Ericsson AB. OTP Design Principles.
- Ericsson AB. Erlang/OTP System Documentation Version 10.7, March 2020.
- Frederic Trottier-Hebert. Learn You Some Erlang For Great Good.
- Ulf Wiger. Four-fold Increase in Productivity and Quality, March 2001.

50