Starting with Erlang

Sequential Programming in Erlang (Overview)

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Erlang
A Few of History

30+ Years

1981 — the Ericsson CS Lab has been founded.

1981–1986
- a lot of work to decide which paradigm would be better to use in the telecommunication domain;
- conclusions: doesn't exist the perfect paradigm but several characteristics should be mixed.

1987 Erlang is born
- the name is after the Danish mathematician Agner Krarup Erlang but could also mean Ericsson language.

1987–1991
- the JAM («Joe's Abstract Machine») virtual machine (inspired by the Prolog WAM) has been implemented (in C);
- in 1998 it has been replaced by BEAM («Bogdan/Björn's Erlang Abstract Machine»).

1996 — Open Telecom Platform (OTP) has been released.

1998
- Ericsson stops to develop Erlang but not to use it
- Erlang becomes open source
  - since 2006 the BEAM supports multi-core processors.

Erlang is concurrency oriented, i.e., the process is the basic of every computation.

Erlang adopts the actor's model for concurrency with
- asynchronous message exchange;
- non shared memory

Erlang is a dynamically typed functional language.

Erlang supports distribution, fault tolerance and hot-swapping (dynamic SW updating).

My First Erlang Program

Again a Factorial!!!

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

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

The program must be run through the BEAM shell

```
[12:56]cazzola@mangog:~/lp/erlang>erl
Erlang/OTP 24 [erts-12.3.2.6] [source] [64-bit] [smp:16:16] [async-threads:1] [jit]

Eshell V12.3.2.6  (abort with ^G)
1> c(fact).
{ok,fact}
2> fact:fact(7).
5040
3> fact:fact(100).
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
```

Alternatively it could be run as a script via escript or through native compilation via HiPE.

```
1> 10.
10
2> 16#FF.
255
3> $A.
65
4> -12.35e-2.
-0.1235
```

- B#val is used to store numbers in base «B»;
- $char is used for ascii values.

```
1> cazzola@di.unimi.it.
'cazzola@di.unimi.it'
2> 'Walter Cazzola'.
'Walter Cazzola'
3> 'Walter^M
3> Cazzola'.
'Walter\nCazzola'
```

- atoms start with lowercase letter but can contain any character;
- if quoted they can start by uppercase letters.

Sequential Erlang Overview

Tuples and Lists

```
1> {123, "walter", cazzola}.
{123,"walter",cazzola}
2> {}.
{}
3> {abc, {'Walter', 'Cazzola'}, 3.14}.
{abc,{'Walter','Cazzola'},3.14}
4>  {{1,2},3}=={1,{2,3}}.
false
```

- used to store a fixed number of items;
- tuples of any size, type and complexity are allowed.

```
1> [].
[]
2> [1|[]].
[1]
3> [1|[2]].
[1,2]
4> [{1,2},ok,[]].
[{1,2},ok,[]]
5> length([{1,2},ok,[]]).
3
6>  [{1,2},ok,[]]==[{1,2},ok,[]].
true
7> A=[$W,$a,$l,$t,$e,$r], B=[$C,$a,$z,$z,$o,$l,$a].
"Cazzola"
8> A++" "++B.
"Walter Cazzola"
9> A--B.
"Wter"
```

- used to store a variable number of items;
- lists are dynamically sized.

```
1> A = 1.
1
2> A = 2.
** exception error:  no match of right hand side value 2
```

- are just name bindings to values and cannot be modified;
- start with an uppercase letter and _ is an anonymous variable.
- the bindings are created via pattern matching.

```
3> [B|L]=[a,b,c].
[a,b,c]
4> {A,B,L}.
{1,a,[b,c]}
5> {X,X}={B,B}.
{a,a}
6> {Y,Y}={X,b}.
** exception error:  no match of right hand side value a,b
7> 1=A.
1
8> 1=Z.
* 1:  variable 'Z' is unbound
9> {A1, _, [B1|_], {B1}} = {abc, 23, [22,x], {22}}.
{abc,23,[22,x],{22}}
10> A1.
abc
11> B1.
22
```

```
name(pattern_11, pattern_12, ..., pattern_1n) [when guard_1] -> body_1 ;
name(pattern_21, pattern_22, ..., pattern_2n) [when guard_2] -> body_2 ;
                                ...
name(pattern_k1, pattern_k2, ..., pattern_kn) [when guard_k] -> body_k .
```

- clauses are scanned sequentially until a match is found;
- when a match is found all the variables in the head become bound;

```
-module(ex_module).
-export([double/1]).

double(X) -> times(X, 2).
times(X, N) -> X * N.
```

- double can be called from outside the module, times is local to the module;
- double/1 means the function double with one argument (note that double/1 and double/2 are two different functions).

Sequential Erlang Overview
Guard Sequences

Each clause in function definition can be guarded by a **guard sequence**.

- a guard is a sequence $G_1, G_2, \ldots, G_n$ of guard expressions;
- a guard expression is a subset of Erlang expressions to guarantee to be free of side-effects;
- a guard sequence is true when all the guard expressions evaluate to true.

Valid guard expression are:

- the atom `true` and other constants;
- calls to some built-in functions (BIFs);
- arithmetic and boolean expressions; and
- short-circuit expressions (`andalso`/`orelse`).

Permitted BIFs are:

is_atom/1	is_binary/1	is_bitstring/1	is_float/1	is_function/2
is_function/1	is_integer/1	is_list/1	is_number/1	is_pid/1
is_port/1	is_record/2	is_record/3	is_reference/1	is_tuple/1
abs/1	bit_size/1	byte_size	element/2	float/1
hd/1	length/1	node/0	node/1	round/1
self/1	size/1	tl/1	trunc/1	tuple_size/1

```erlang
-module(mfr).
-export([map/2,filter/2,reduce/2]).

map(_, []) -> [];
map(F, [H|TL]) -> [F(H)|map(F,TL)].

filter(_, []) -> [];
filter(P, [H|TL]) -> filter(P(H), P, H, TL).

filter(true, P, H, L) -> [H|filter(P, L)];
filter(false, P, _, L) -> filter(P, L).

reduce(F, [H|TL]) -> reduce(F, H, TL).

reduce(_, Q, []) -> Q;
reduce(F, Q, [H|TL]) -> reduce(F, F(Q,H), TL).
```

```
1> mfr:map(fun(X) -> X*X end, [1,2,3,4,5,6,7]).
[1,4,9,16,25,36,49]
2> mfr:filter(fun(X) -> (X rem 2)==0 end, [1,2,3,4,5,6,7]).
[2,4,6]
3> mfr:reduce(fun(X,Y) -> X+Y end, [1,2,3,4,5,6,7]).
28
```

They are available in the module lists.

```
[X||Qualifier₁, ..., Qualifier_n]
```

X is an expression, each qualifier is a generator or a filter

- generators are in the form `Pattern <- ListExpr` where `ListExpr` evaluates to a list;
- filters are either predicates or boolean expressions.

```
-module(sort).
-export([qsort/2]).

qsort(_, []) -> [];
qsort(P, [Pivot|TL]) ->
  qsort(P, [X||X<-TL, P(X,Pivot)]) ++ [Pivot] ++  qsort(P, [X||X<-TL, not P(X,Pivot)]).
```

```
-module(prime).
-export([primes/1]).

primes(N) when N>1 -> [X|| X <- lists:seq(2,N),
   (length([Y || Y <- lists:seq(2, trunc(math:sqrt(X))), ((X rem Y) == 0)]) == 0)];
primes(_) -> [].
```

```
1> sort:qsort(fun(X,Y) -> X<Y end, [13,1,-1,8,9,0,3.14]).
[-1,0,1,3.14,8,9,13]
2> sort:qsort(fun(X,Y) -> X>Y end, [13,1,-1,8,9,0,3.14]).
[13,9,8,3.14,1,0,-1]

3> prime:primes(100).
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

Threads are the traditional way of offering concurrency

- the execution of the program is split up into concurrently running tasks;
- such tasks operate on shared memory

Several problems

- race conditions with update loss

T_1 (`withdraw(10)`)	T_2 (`withdraw(10)`)	Balance
`if (balance - amount >= 0)`		15 €
	`if (balance - amount >= 0)`	15 €
	`balance -= amount;`	5 €
`balance -= amount;`		-5 €

- deadlocks

P_1	P_2
`lock(A)`	`lock(B)`
`lock(B)`	`lock(A)`

Erlang (and also Scala via the Akka library) takes a different approach to concurrency: the Actor Model.

Each object is an actor.

- it has a mailbox and a behavior;
- actors communicate through messages buffered in a mailbox.

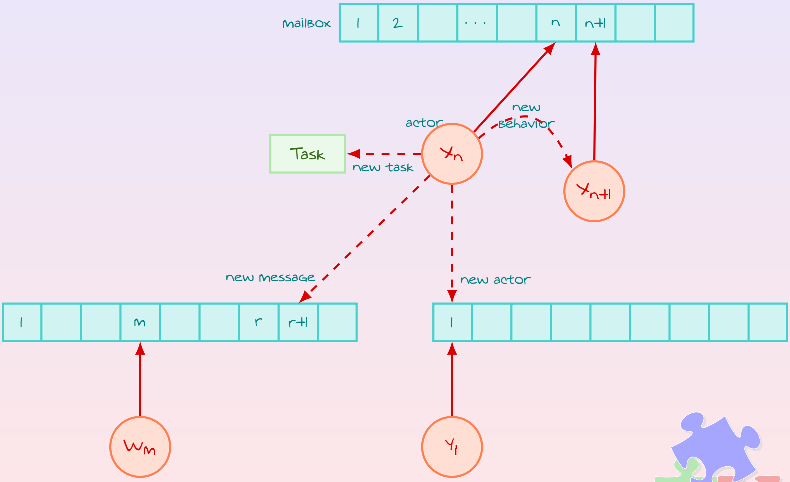
Computation is data-driven, upon receiving a message an actor

- can send a number of messages to other actors;
- can create a number of actors; and
- can assume a different behavior for dealing with the next message in its mailbox.

Note that,

- all communications are performed asynchronously;
  - the sender does not wait for a message to be received upon sending it;
  - no guarantees about the receiving order but they will eventually be delivered.
- there is no shared state between actors
  - information about internal state are requested/provided by messages;
  - also internal state manipulation happens through messages.
- actors run concurrently and are implemented as lightweight user-space threads

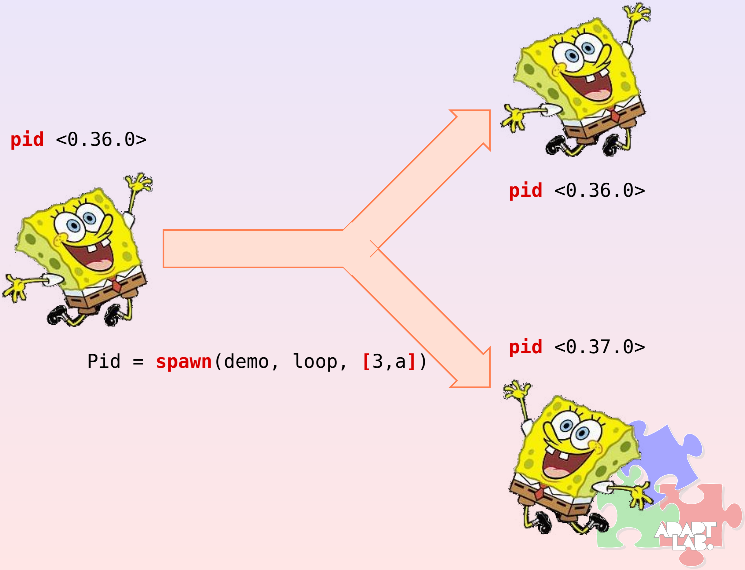
Actor Model Concurrency
Transaction Overview



Three basic elements form the foundation for concurrency

- a built-in function (`spawn()`) to create new actors;
- an operator (`!`) to send a message to another actor; and
- a mechanism to pattern-match message from the actor's mailbox.

Concurrency in Erlang
Spawning New Processes.



```erlang
-module(processes_demo).
-export([start/2, loop/2]).

start(N,A) -> spawn (processes_demo, loop, [N,A]).

loop(0,A) -> io:format("~p(~p) ~p~n", [A, self(), stops]);
loop(N,A) -> io:format("~p(~p) ~p~n", [A, self(), N]), loop(N-1,A).
```

```
1> processes_demo:start(7,a),processes_demo:start(5,b),processes_demo:start(3,c).
a(<0.73.0>) 7
b(<0.74.0>) 5
a(<0.73.0>) 6
c(<0.75.0>) 3
b(<0.74.0>) 4
<0.75.0>
a(<0.73.0>) 5
c(<0.75.0>) 2
b(<0.74.0>) 3
a(<0.73.0>) 4
c(<0.75.0>) 1
b(<0.74.0>) 2
a(<0.73.0>) 3
c(<0.75.0>) stops
b(<0.74.0>) 1
a(<0.73.0>) 2
b(<0.74.0>) stops
a(<0.73.0>) 1
a(<0.73.0>) stops
```

`self()` returns the PID of the process.

Every actor is characterized by:

- an address which identifies the actor and
- a mailbox where the delivered messages but not cleared yet are stored;

Messages are sorted on arrival time (not on sending time).

To send a message to an actor:

- has to know the address (pid) of the target actor;
- to send its address (pid) to the target with the message if a reply is necessary; and
- to use the send (!) primitive.

Exp_1 ! Exp_2

- Exp_1 must identify an actor;
- Exp_2 any valid Erlang expression; the result of the send expression is the one of Exp_2;
- the sending never fails also when the target actor doesn't exist or is unreachable;
- the sending operation never block the sender.

The receiving operation uses pattern matching.

```
receive
  Any -> do_something(Any)
end
```

- the actor pick out of the mailbox the oldest message matching Any;
- it is blocked waiting for a message when the queue is empty.

```
receive
  {Pid, something} -> do_something(Pid)
end
```

- the actor tries to pick out the oldest message that matches {Pid, something};
- if it fails the actor is blocked waiting for such a message

```
receive
  Pattern1 [when GuardSeq1] -> Body1 ;
                  ...
  Patternn [when GuardSeqn] -> Bodyn
[after Exprt -> Bodyt]
end
```

- rules definition and evaluation is quite similar to the functions;
- when no pattern matches the mailbox the actor waits instead of raising an exception;
- to avoid waiting forever the clause after can be used, after $Expr_t$ ms the actor is woke up.

Concurrency in Erlang
Converting Some Temperatures.

```erlang
-module(converter).
-export([t_converter/0]).

t_converter() ->
 receive
   {toF, C} -> io:format("~p °C is ~p °F~n", [C, 32+C*9/5]), t_converter();
   {toC, F} -> io:format("~p °F is ~p °C~n", [F, (F-32)*5/9]), t_converter();
   {stop} -> io:format("Stopping!~n");
   Other -> io:format("Unknown: ~p~n", [Other]), t_converter()
 end.
```

```
1> Pid = spawn(converter, t_converter, []).
<0.39.0>
2> Pid ! {toC, 32}.
32 °F is 0.0 °C
{toC,32}
3> Pid ! {toF, 100}.
100 °C is 212.0 °F
{toF,100}
4> Pid ! {stop}.
Stopping!
{stop}
5> Pid ! {toF, 100}. % once stopped a message to such a process is silently ignored
{toF,100}
```

Concurrency in Erlang
Calculating Some Areas.

```erlang
-module(area_server).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop();
        {circle, R} ->
            io:format("Area of circle is ~p~n", [3.14159 * R * R]),
            loop();
        Other ->
            io:format("I don't know how to react to the message ~p~n",[Other]),
            loop()
    end.
```

```
1> Pid = spawn(fun area_server:loop/0).
<0.34.0>
2> Pid ! {rectangle, 30, 40}.
Area of rectangle is 1200
{rectangle,30,40}
4> Pid ! {circle, 40}.
Area of circle is 5026.544
{circle,40}
5> Pid ! {triangle,22,44}.
I don't know what the area of a {triangle,22,44} is
{triangle,22,44}
```

Actors are not processes and are not dealt by the operating system

- the BEAM uses a preemptive scheduler;
- when an actor run for a too long period of time or when it enters a `receive` statement with no message available, the actor is halted and placed on a scheduling queue;

Actors and the rest of the system

- OS processes and actors have different schedulers and long running Erlang applications do not interfere with the execution of the OS processes (no one will become unresponsive)
- the BEAM supports symmetric multiprocessing (SMP)
  - i.e., it can run processes in parallel on multiple CPUs
  - but it cannot run lightweight processes (actors) in parallel on multiple CPUs.

Timing the Spawning Process.


```erlang
-module(processes).
-export([max/1]).

max(N) ->
  Max = erlang:system_info(process_limit),
  io:format("Maximum allowed processes:~p~n",[Max]),
  statistics(runtime), statistics(wall_clock),
  L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
  {_, Time1} = statistics(runtime), {_, Time2} = statistics(wall_clock),
  lists:foreach(fun(Pid) -> Pid ! die end, L),
  U1 = Time1 * 1000 / N, U2 = Time2 * 1000 / N,
  io:format("Process spawn time = ~p (~p) microseconds~n", [U1, U2]).

wait() -> receive die -> void end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

```
1> processes:max(20000).
Maximum allowed processes:32768
Process spawn time = 2.5 (3.4) microseconds
ok
2> processes:max(40000).
Maximum allowed processes:32768

=ERROR REPORT==== 8-Nov-2011::14:24:32 ===
Too many processes
...
[16:48]cazzola@surtur:~/lp/erlang>erl +P 100000
1> processes:max(50000).
Maximum allowed processes:100000
Process spawn time = 3.2 (3.74) microseconds
ok
```

Erlang provides a mechanism to render public the pid of a process to all the other processes.

- register(an_atom, Pid)
- unregister(an_atom)
- whereis(an_atom)->Pid|undefined
- registered()

Once registered

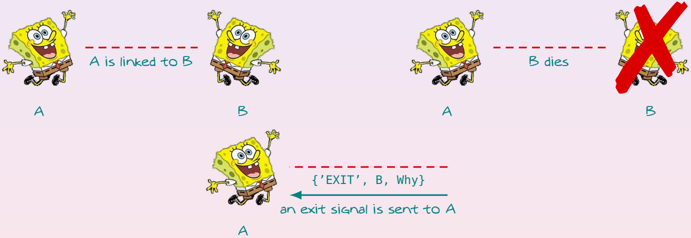
- it is possible to send a message to it directly (name!msg).

```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) -> register(clock, spawn(fun() -> tick(Time, Fun) end)).
stop() -> clock ! stop.

tick(Time, Fun) ->
  receive
    stop -> void
  after
    Time -> Fun(), tick(Time, Fun)
  end.
```

```
5> clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).
true
TICK 1320,769016,673190
TICK 1320,769021,678451
TICK 1320,769026,679120
7> clock:stop().
stop
```

When two processes are related

- the errors of one affect the behavior of the other process;
- the BIF `link` function helps to monitor.



If A is linked to B

- when B dies an exit signal is sent to A;
- the signal is a message like `{'EXIT', Pid, _}`.

```
-module(dies).
-export([on_exit/2]).

on_exit(Pid, Fun) ->
  spawn(fun() ->
      process_flag(trap_exit, true),
      link(Pid),
      receive
        {'EXIT', Pid, Why} -> Fun(Why)
      end
  end).
```

```
1> F = fun() -> receive X -> list_to_atom(X) end end.
#Fun<erl_eval.20.67289768>
2> Pid = spawn(F).
<0.35.0>
3> dies:on_exit(Pid, fun(Why) -> io:format("~p died with:~p~n",[Pid, Why]) end).
<0.37.0>
4> Pid ! hello.
<0.35.0> died with:{badarg,[{erlang,list_to_atom,[hello]}]}

=ERROR REPORT==== 9-Nov-2011::17:50:20 ===
Error in process <0.35.0> with exit value:  badarg,[{erlang,list_to_atom,[hello]}]}
hello
```

Links

- defines an error propagation path between two processes;
- if a process dies an exit signal is sent to the other process;
- the set of processes linked to a given process is called link set.

Exit Signals

- they are generated by a process when it dies;
- signals are broadcast to all processes in the link set of the dying process;
- the exit signal contains an argument explaining why the process died (`exit(Reason)` or implicitly set).
- when a process «naturally dies» the exit reason is `normal`;
- exit signals can be explicitly sent via `exit(Pid, X)`: the sender does not die («fake death»).

System Processes

- a non system process that receives a exit signal dies too;
- a system process receives the signal as a normal message in its mailbox;
- `process_flag(trap_exit, true)` transform a process into a system process.

Receiver's Behavior

trap_exit	Exit Signal	Action
true	kill	dies & broadcasts it to its link set
true	X	adds {'EXIT', Pid, X} to the mailbox
false	normal	continues & the signal vanishes
false	kill	dies & broadcasts it to its link set
false	X	dies & broadcasts it to its link set

Privileged (System process)

Alternatives

- I don't care if a process I create crashes.
```
Pid = spawn(fun() ->... end)
```
- I want to die if a process I create crashes.
```
Pid = spawn_link(fun() ->... end)
```
- I want to handle errors if a process I create crashes
```
process_flag(trap_exits, true),
Pid = spawn_link(fun() ->... end).
```

```
-module(edemo1).
-export([start/2]).

start(Bool, M) ->
  A = spawn(fun() -> a() end),
  B = spawn(fun() -> b(A, Bool) end),
  C = spawn(fun() -> c(B, M) end),
  sleep(1000), status(b, B), status(c, C).

a() -> process_flag(trap_exit, true), wait(a).
b(A, Bool) -> process_flag(trap_exit, Bool), link(A), wait(b).
c(B, M) -> link(B),
  case M of
    {die, Reason} -> exit(Reason);
    {divide, N} -> 1/N, wait(c);
    normal -> true
  end.
```

This starts 3 processes: A, B and C

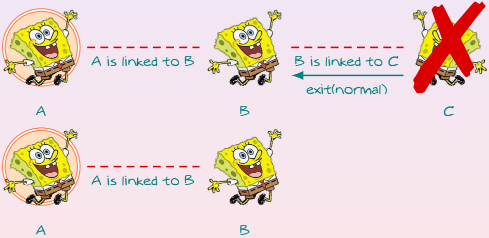
- A will trap exits and watch for exits from B;
- B will trap exits if Bool is true and
- C will die with exit reason M.

```
wait(Prog) ->
  receive
    Any ->
      io:format("Process ~p received ~p~n", [Prog, Any]),
      wait(Prog)
  end.

sleep(T) ->
  receive
  after T -> true
  end.

status(Name, Pid) ->
  case erlang:is_process_alive(Pid) of
    true -> io:format("process ~p (~p) is alive~n", [Name, Pid]);
    false -> io:format("process ~p (~p) is dead~n", [Name, Pid])
  end.
```

This starts 3 processes: A, B and C

- `wait/1` just prints any message it receives;
- `sleep/1` awakes the invoking process after a period of time;
- `status/2` prints the aliveness of the invoing process.

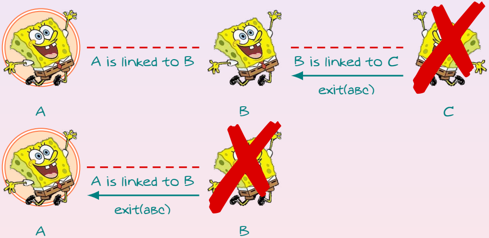
```
1> edemo1:start(false, {die,normal}).
process b (<0.48.0>) is alive
process c (<0.49.0>) is dead
ok
```



- B is not a system process;
- when C dies with `normal` signal, B doesn't die.

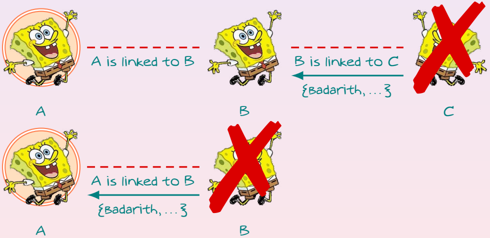
Errors in Concurrent Programs
Going into Details of Error Handling (Cont'd)

```
1> edemo1:start(false, {die, abc}).
Process a received {'EXIT',<0.40.0>,abc}
process b (<0.40.0>) is dead
process c (<0.41.0>) is dead
ok
```



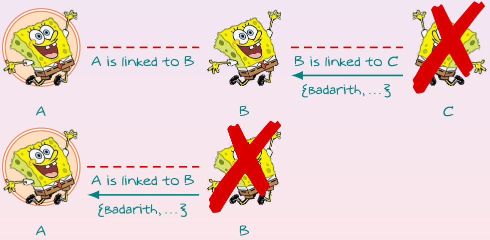
- B is not a system process;
- when C evaluates **exit**(abc), process B dies;
- when B exits rebroadcasts the unmodified exit signal to its link set
- A traps the exit signal and convert it to the error message

```
6> edemo1:start(false, {divide,0}).
Process a received {'EXIT',<0.56.0>,{badarith,[{edemo1,c,2}]}}
=ERROR REPORT==== 11-Nov-2011::18:03:29 ===
Error in process <0.57.0> with exit value: {badarith,[{edemo1,c,2}]}
process b (<0.56.0>) is dead
process c (<0.57.0>) is dead
ok
```



- B is not a system process;
- when C tries to divide by zero an error occurs and C dies with a `{badarith, ...}` error;
- B receives this and dies and the error is propagated to A.

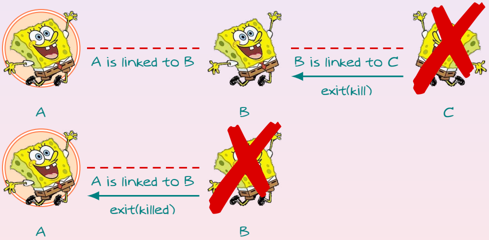
```
6> edemo1:start(false, {divide,0}).
Process a received {'EXIT',<0.56.0>,{badarith,[{edemo1,c,2}]}}
=ERROR REPORT==== 11-Nov-2011::18:03:29 ===
Error in process <0.57.0> with exit value: {badarith,[{edemo1,c,2}]}
process b (<0.56.0>) is dead
process c (<0.57.0>) is dead
ok
```



- B is not a system process;
- when C tries to divide by zero an error occurs and C dies with a `{badarith, ...}` error;
- B receives this and dies and the error is propagated to A.

Errors in Concurrent Programs
Going into Details of Error Handling (Cont'd)

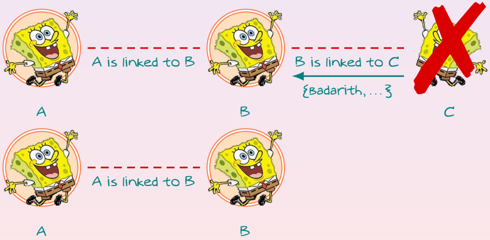
```
1> edemo1:start(false, {die, kill}).
Process a received {'EXIT',<0.60.0>,killed}
process b (<0.60.0>) is dead
process c (<0.61.0>) is dead
ok
```



- B is not a system process;
- the exit reason kill causes B to die, and the error is propagated to its link set.

Errors in Concurrent Programs
Going into Details of Error Handling (Cont'd)

```
8> edemo1:start(true, {divide,0}).
Process b received {'EXIT',<0.65.0>,{badarith,[{edemo1,c,2}]}}
=ERROR REPORT==== 11-Nov-2011::18:16:47 ===
Error in process <0.65.0> with exit value: {badarith,[{edemo1,c,2}]}
process b (<0.64.0>) is alive
process c (<0.65.0>) is dead
ok
```



- B is a system process;
- in all cases, B traps the error;
- the error is never propagated to A.

Links are symmetric

- i.e., if A dies, B will sent an exit signal and vice versa;
- to prevent a process from dying, we have to make it a system process that is not alway desirable.

A monitor is an asymmetric link

- if A monitors B and B dies A will be sent an exit signal but
- if A dies B will not be sent a signal.

A can create a monitor for B calling `erlang:monitor(process, B)`

- if B dies with exit reason `Reason` a `'DOWN'` message

  ```
  {'DOWN', Ref, process, B, Reason}
  ```

  is sent to A (`Ref` is the reference to the monitor).
- the monitor is unidirectional:
  - to repeat the above call will create several, independent monitors and each one will send a `'DOWN'` message when B terminates.

Performance

- to speed up programs by arranging that different parts of the program are run in parallel on different machines.

Reliability

- to make fault tolerant systems by structuring the system to be replicated on several machines: if one fails the computation continues on another machine.

Scalability

- resources on a single machine tend to be exhausted;
- to add another computer means to double the resources.

Intrinsically Distributed Applications

- e.g., chat systems, multi-user games, ...

Erlang provides two models of distribution: distributed Erlang and socket based distribution

Distributed Erlang

- applications run on a set of tightly coupled computers called Erlang nodes;
- processes can be spawned on every node, and
- apart from the spawning all things still work as always

Socket-Based Distribution

- it can run in an untrusted environment;
- less powerful (restricted connections);
- fine grained control on what can be executed on a node.

```
-module(kvs).
-export([start/0, store/2, lookup/1]).

start() -> register(kvs, spawn(fun() -> loop() end)).
store(Key, Value) -> rpc({store, Key, Value}).
lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
  kvs ! {self(), Q},
  receive
    {kvs, Reply} -> Reply
  end.

loop() ->
  receive
    {From, {store, Key, Value}} -> put(Key, {ok, Value}), From ! {kvs, true}, loop();
    {From, {lookup, Key}} -> From ! {kvs, get(Key)}, loop()
  end.
```

The name server reply to the protocol

- `start()` that starts the server with the registered name kvs;
- `lookup(Key)` returns the value associated to the Key into the name server; and
- `store(Key, Value)` associate the Value to the Key into the name server.

Distributed Programming in Erlang
Our First Distributed Program: a Name Server (Cont'd)

Sequential Execution

```
1> kvs:start().
true
2> kvs:store({location, walter}, "Genova").
true
3> kvs:store(weather, sunny).
true
4> kvs:lookup(weather).
{ok,sunny}
5> kvs:lookup({location, walter}).
{ok,"Genova"}
6> kvs:lookup({location, cazzola}).
undefined
```

Sullo stesso nodo della VM di Erlang

Distributed but on Localhost

```
[15:58]cazzola@surtur:~/lp/erlang>erl -sname sif
(sif@surtur)1> kvs:start().
true

(sif@surtur)2> kvs:lookup(weather).
{ok,sunny}
```

```
[15:58]cazzola@surtur:~/lp/erlang>erl -sname amora
(amora@surtur)1>
   rpc:call(sif@surtur, kvs, store, [weather, sunny]).
true
(amora@surtur)2>
   rpc:call(sif@surtur, kvs, lookup, [weather]).
{ok,sunny}
```

user fun fun:fun dict

Distributed on two separate computers (surtur and thor)

```
[16:31]cazzola@surtur:~/lp/erlang> ssh thor
[16:32]cazzola@thor:~>erl -name sif -setcookie abc
(sif@thor)1> kvs:start().
true
(sif@thor)2> kvs:lookup(weather).
{ok,warm}
```

```
[16:32]cazzola@surtur:>erl -name amora -setcookie ab
(amora@surtur)1>
   rpc:call(sif@thor, kvs, store, [weather, warm]).
true
(amora@surtur)2>
   rpc:call(sif@thor, kvs, lookup, [weather]).
{ok,warm}
```

Node is the central concept.

- it is a self-contained Erlang system VM with its own address space and own set of processes;
- the access to a single node is secured by a cookie system
  - each node has a cookie and
  - it must be the same of any node to which the node talks;
  - the cookie is set when the VM starts or using `erlang:set_cookie`.
- the set of nodes with the same cookie define a cluster

Primitives for writing distributed programs are:

- `spawn(Node, Mod, Func, ArgList)->Pid`
- `spawn_link(Node, Mod, Func, ArgList)->Pid`
- `disconnect_node(Node)->bools() | ignored`
- `monitor_node(Node, Flag)->true`
- `{RegName, Node}!Msg`

```erlang
-module(ddemo).
-export([rpc/4, start/1]).

start(Node) -> spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->
  Pid ! {rpc, self(), M, F, A},
  receive
    {Pid, Response} -> Response
  end.

loop() ->
  receive
    {rpc, Pid, M, F, A} ->
       Pid ! {self(), (catch apply(M, F, A))},
       loop()
  end.
```

```
[19:01]cazzola@surtur:~/lp/erlang>erl -name sif -setcookie abc
(sif@surtur.di.unimi.it)1> Pid = ddemo:start('amora@thor.di.unimi.it').
<8745.43.0>
(sif@surtur.di.unimi.it)3> ddemo:rpc(Pid, erlang, node, []).
'amora@thor.di.unimi.it'
```

Macchina virtuale diversa -> Nodo =/= 0

Note

- Erlang provides specific libraries with support for distribution look at: rpc and global.

Two nodes to communicate MUST have the same magic cookie.

Three ways to set the cookie:

1. to store the cookie in `$HOME/.erlang.cookie`

```
[19:26]cazzola@surtur:~/lp/erlang>echo "A Magic Cookie" > ~/.erlang.cookie
[19:27]cazzola@surtur:~/lp/erlang>chmod 400 ~/.erlang.cookie
```

2. through the option `-setcookie`

```
[19:27]cazzola@surtur:~/lp/erlang>erl -setcookie "A Magic Cookie"
```

3. by using the BIF `erlang:set_cookies`

```
[19:34]cazzola@surtur:~/lp/erlang>erl -sname sif
(sif@surtur)1> erlang:set_cookie(node(), 'A Magic Cookie').
true
```

Note that 1 and 3 are safer than 2 and the cookies never wander on the net in clear.

Problem with spawn-based distribution

- the client can spawn any process on the server machine
- e.g., `rpc:multicall(nodes(), os, cmd, ["cd /; rm -rf *"])`

Spawn-based distribution

- is perfect when you own all the machines and you want to control them from a single machine; but
- is not suited when different people own the machines and want to control what is in execution on their machines.

Socket-base distribution

- will use a restricted form of spawn where the owner of a machine has explicit control over what is run on his machine;
- `lib_chan`;

lib_chan is a module

- that allows a user to explicitly control which processes are spawned on his machines.

The interface is as follows

- `start_server()->true`
  this starts a server on local host, whose behavior depends on `$HOME/.erlang_config/lib_chan.conf`
- `connect(Host, Port, S, P, ArgsC)->{ok, Pid}|{error, Why}`
  try to open the port `Port` on the host `Host` and then to activate the service `S` protected by the password `P`.

The configuration file contains tuples of the form:

- `{port, NNNN}`
  this starts listening to port number `NNNN`
- `{service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgs}`
  - this defines a service `S` protected by password `P`;
- When the connection is created by the `connect` call, the server spawns

  `SomeMod:SomeFunc(MM, ArgC, SomeArgs)`

  - where `MM` is the `Pid` of a proxy process to send messages to the client and `ArgsC` comes from the client `connect` call.

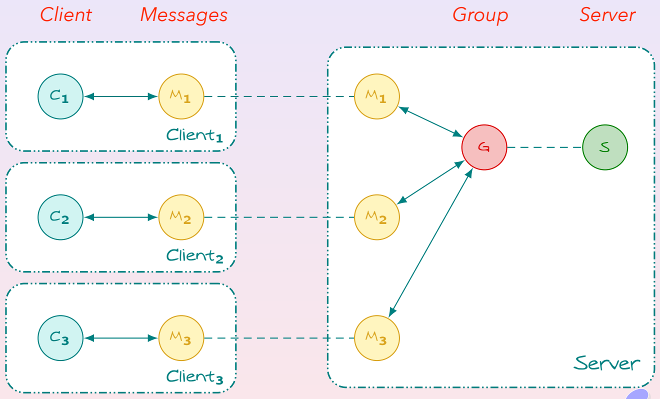
```
{port, 12340}.
{service, nameServer, password, "ABXy45", mfa, mod_name_server, start_me_up, notUsed}.
```

```
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgS) -> loop(MM).

loop(MM) ->
  receive
    {chan, MM, {store, K, V}} -> kvs:store(K,V), loop(MM);
    {chan, MM, {lookup, K}} -> MM ! {send, kvs:lookup(K)}, loop(MM);
    {chan_closed, MM} -> true
  end.
```

```
1> kvs:start().
true
2> lib_chan:start_server().
Starting a port server on 12340...
true
3> kvs:lookup(joe).
{ok,"writing a book"}
```

```
1> {ok, Pid} = lib_chan:connect("localhost", 12340, nameServer, "ABXy45", "").
{ok, <0.43.0>}
2> lib_chan:cast(Pid, {store, joe, "writing a book"}).
{send,{store,joe,"writing a book"}}
3> lib_chan:rpc(Pid, {lookup, joe}).
{ok,"writing a book"}
4> lib_chan:rpc(Pid, {lookup, jim}).
undefined
```

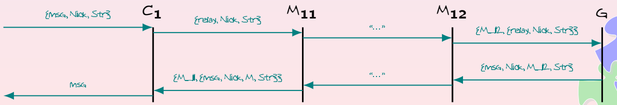
IRC lite
The Architecture



IRC lite
The Architecture (Cont'd)

The IRC-lite system is composed of
- 3 client nodes running on different machines and
- a single server node on another machine.

Such components perform the following functions:
- the chat clients send/receive messages to/from the group control;
- the group controller manages a single chat group;
  - a message sent to the controller is broadcast to all the group members
- the chat server tracks the group controllers and manages the joining operation; and
- the middle-men take care of the transport of data (they hide the sockets).



```erlang
-module(chat_client).
-export([start/1,connect/5]).

start(Nick) -> connect("localhost", 2223, "AsDT67aQ", "general", Nick).
```

```erlang
connect(Host, Port, HostPsw, Group, Nick) ->
   spawn(fun() -> handler(Host, Port, HostPsw, Group, Nick) end).

handler(Host, Port, HostPsw, Group, Nick) ->
  process_flag(trap_exit, true),
  start_connector(Host, Port, HostPsw),
  disconnected(Group, Nick).
```

- it makes itself into a system process;
- it then spawns a connection process (which tries to connect to the server);
- it waits for a connection event in disconnected.

```erlang
disconnected(Group, Nick) ->
  receive
    {connected, MM} ->                    % from the connection process
       io:format("connected to server\nsending data\n"),
       lib_chan_mm:send(MM, {login, Group, Nick}),
       wait_login_response(MM);
    {status, S} -> io:format("~p~n",[S]), disconnected(Group, Nick);
    Other ->
       io:format("chat_client disconnected unexpected:~p~n",[Other]),
       disconnected(Group, Nick)
  end.
```

IRC lite
The Client Implementation (Cont'd).

```
start_connector(Host, Port, Pwd) ->
   S = self(), spawn_link(fun() -> try_to_connect(S, Host, Port, Pwd) end).
```

Note that

```
S=self(), spawn_link(fun() -> try_to_connect(S, ...) end)
```

is different than

```
spawn_link(fun() -> try_to_connect(self(), ...) end)
```

```
try_to_connect(Parent, Host, Port, Pwd) ->
  %% Parent is the Pid of the process that spawned this process
  case lib_chan:connect(Host, Port, chat, Pwd, []) of
    {error, _Why} ->
       Parent ! {status, {cannot, connect, Host, Port}},
       sleep(2000),
       try_to_connect(Parent, Host, Port, Pwd);
    {ok, MM} ->
       lib_chan_mm:controller(MM, Parent),
       Parent ! {connected, MM}, %% to disconnected
       exit(connectorFinished)
  end.
sleep(T) -> receive after T -> true end.
```

The Client Implementation (Cont'd).


```
wait_login_response(MM) ->
  receive
    {chan, MM, ack} -> active(MM);
    {'EXIT', _Pid, connectorFinished} -> wait_login_response(MM);
    Other ->
       io:format("chat_client login unexpected:~p~n",[Other]),
       wait_login_response(MM)
  end.
```

```
active(MM) ->
  receive
    {msg, Nick, Str} ->
        lib_chan_mm:send(MM, {relay, Nick, Str}),
        active(MM);
    {chan, MM, {msg, From, Pid, Str}} ->
        io:format("~p@~p: ~p~n", [From,Pid,Str]),
        active(MM);
    {close, MM} -> exit(serverDied);
    Other ->
        io:format("chat_client active unexpected:~p~n",[Other]),
        active(MM)
   end.
```

`active`

- sends messages to the group and vice versa and
- monitors the connection with the group

```
{port, 2223}.
{service, chat, password,"AsDT67aQ",mfa,chat_controller,start,[]}.
```

– it uses lib_chan.

```
-module(chat_controller).
-export([start/3]).
-import(lib_chan_mm, [send/2]).

start(MM, _, _) ->
  process_flag(trap_exit, true),
  io:format("chat_controller off we go ...~p~n",[MM]),
  loop(MM).

loop(MM) ->
  receive
    {chan, MM, Msg} ->                                  %% when a client connects
        chat_server ! {mm, MM, Msg},
        loop(MM);
    {'EXIT', MM, _Why} ->                           %% when the session terminates
        chat_server ! {mm_closed, MM};
    Other ->
        io:format("chat_controller unexpected message =~p (MM=~p)~n", [Other, MM]),
        loop(MM)
  end.
```

IRC lite
The Server Implementation: The Chat Server.


```
-module(chat_server).

start() -> start_server(), lib_chan:start_server("chat.conf").

start_server() ->
  register(chat_server,
    spawn(fun() ->
      process_flag(trap_exit, true),
      Val = (catch server_loop([])),
      io:format("Server terminated with:~p~n",[Val])
    end)).

server_loop(L) ->
  receive
    {mm, Channel, {login, Group, Nick}} ->
       case lookup(Group, L) of
         {ok, Pid} -> Pid ! {login, Channel, Nick}, server_loop(L);
         error ->
            Pid = spawn_link(fun() -> chat_group:start(Channel, Nick) end),
            server_loop([{Group,Pid}|L])
       end;
    {mm_closed, _} -> server_loop(L);
    {'EXIT', Pid, allGone} -> L1 = remove_group(Pid, L), server_loop(L1);
    Msg -> io:format("Server received Msg=~p~n", [Msg]), server_loop(L)
  end.

lookup(G, [{G,Pid}|_]) -> {ok, Pid};
lookup(G, [_|T])       -> lookup(G, T);
lookup(_,[])           -> error.

remove_group(Pid, [{G,Pid}|T]) -> io:format("~p removed~n",[G]), T;
remove_group(Pid, [H|T])       -> [H|remove_group(Pid, T)];
remove_group(_, [])            -> [].
```

```erlang
-module(chat_group).
-export([start/2]).

start(C, Nick) ->
  process_flag(trap_exit, true),
  lib_chan_mm:controller(C, self()), lib_chan_mm:send(C, ack),
  self() ! {chan, C, {relay, Nick, "I'm starting the group"}},
  group_controller([{C,Nick}]).

delete(Pid, [{Pid,Nick}|T], L) -> {Nick, lists:reverse(T, L)};
delete(Pid, [H|T], L)          -> delete(Pid, T, [H|L]);
delete(_, [], L)               -> {"????", L}.

group_controller([]) -> exit(allGone);
group_controller(L) ->
  receive
    {chan, C, {relay, Nick, Str}} ->
      lists:foreach(fun({Pid,_}) -> lib_chan_mm:send(Pid, {msg,Nick,C,Str}) end, L),
      group_controller(L);
    {login, C, Nick} ->
      lib_chan_mm:controller(C, self()), lib_chan_mm:send(C, ack),
      self() ! {chan, C, {relay, Nick, "I'm joining the group"}},
      group_controller([{C,Nick}|L]);
    {chan_closed, C} ->
      {Nick, L1} = delete(C, L, []),
      self() ! {chan, C, {relay, Nick, "I'm leaving the group"}},
      group_controller(L1);
    Any ->
      io:format("group controller received Msg=~p~n", [Any]),
      group_controller(L)
    end.
```

IRC lite
Chatting around . . .


```
1> chat_server:start().
lib_chan starting:"chat.conf"
ConfigData=[{port,2223}, {service,chat,password,"AsDT67aQ",mfa,chat_controller,start,[]}]
chat_controller off we go ...<0.39.0>
chat_controller off we go ...<0.41.0>
chat_controller off we go ...<0.43.0>
server error should die with exit(normal) was:{mm_closed,<0.39.0>}
chat_controller off we go ...<0.46.0>
server error should die with exit(normal) was:mm_closed,<0.46.0>}
server error should die with exit(normal) was:mm_closed,<0.41.0>}
server error should die with exit(normal) was:mm_closed,<0.43.0>}
```

```
1> ChatDaemon = chat_client:start(walter).
walter@<0.41.0>: "I'm joining the group"
'walter cazzola'@<0.43.0>: "I'm joining the group"
2> ChatDaemon ! {msg, walter, "Hello World!!!"}.
{msg,walter,"Hello World!!!"}
walter@<0.41.0>: "Hello World!!!"
'walter cazzola'@<0.43.0>: "Hello Walter!!!"
cazzola@<0.39.0>: "Hello Walter!!!"
cazzola@<0.39.0>: "I'm leaving the group"
cazzola@<0.46.0>: "I'm joining the group"
cazzola@<0.46.0>: "I'm leaving the group"
```

```
1> ChatDaemon = chat_client:start('walter cazzola').
'walter cazzola'@<0.43.0>: "I'm joining the group"
walter@<0.41.0>: "Hello World!!!"
2> ChatDaemon!{msg,'walter cazzola',"Hello Walter!!!"}.
{msg,'walter cazzola',"Hello Walter!!!"}
'walter cazzola'@<0.43.0>: "Hello Walter!!!"
cazzola@<0.39.0>: "Hello Walter!!!"
cazzola@<0.39.0>: "I'm leaving the group"
cazzola@<0.46.0>: "I'm joining the group"
cazzola@<0.46.0>: "I'm leaving the group"
walter@<0.41.0>: "I'm leaving the group"
```

```
1> ChatDaemon = chat_client:start(cazzola).
cazzola@<0.39.0>: "I'm starting the group"
walter@<0.41.0>: "I'm joining the group"
'walter cazzola'@<0.43.0>: "I'm joining the group"
walter@<0.41.0>: "Hello World!!!"
'walter cazzola'@<0.43.0>: "Hello Walter!!!"
2> ChatDaemon ! {msg, cazzola, "Hello Walter!!!"}.
{msg,cazzola,"Hello Walter!!!"}
cazzola@<0.39.0>: "Hello Walter!!!"
3> ^C [21:35]cazzola@surtur:~/lp/erlang/chat>erl
1> ChatDaemon = chat_client:start(cazzola).
cazzola@<0.46.0>: "I'm joining the group"
```

References

- Gul Agha.
  Actors: A Model of Concurrent Computation in Distributed Systems.
  MIT Press, Cambridge, 1986.
- Joe Armstrong.
  Programming Erlang: Software for a Concurrent World.
  The Pragmatic Bookshelf, fifth edition, 2007.
- Francesco Cesarini and Simon Thompson.
  Erlang Programming: A Concurrent Approach to Software Development.
  O'Reilly, June 2009.