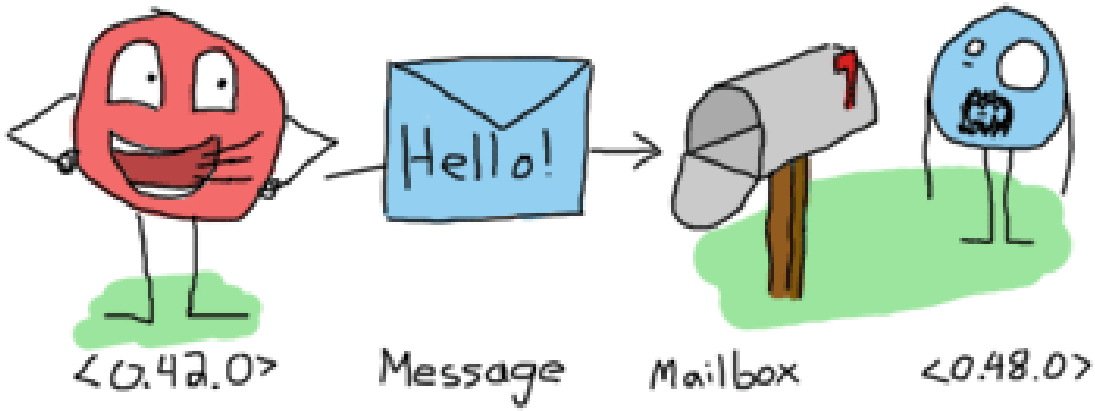
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean



http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency#dont-panic

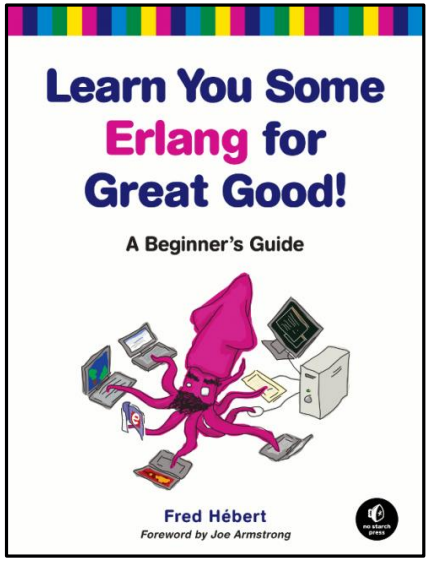
ACTOR MODEL

"Erlang's actor model can be imagined as a world where everyone is sitting alone in their own room and can perform a few distinct tasks. Everyone communicates strictly by writing letters and that's it. While it sounds like a boring life (and a new age for the postal service), it means you can ask many people to perform very specific tasks for you, and none of them will ever do something wrong or make mistakes which will have repercussions on the work of others; they may not even know the existence of people other than you (and that's great).

To escape this analogy, Erlang forces you to write actors (processes) that will share no information with other bits of code unless they pass messages to each other. Every communication is explicit, traceable and safe."

Fred Hébert, Learn You Some Erlang For Great Good

http://learnyousomeerlang.com/introduction#what-is-erlang



Varianta online

➢Bibliografie

Joe Armstrong, Programming Erlang, Second Edition 2013

Fred Hébert, Learn You Some Erlang For Great Good, 2013
Varianta online

Jim Larson, Erlang for Concurrent Programming, ACM Queue, 2008

CONCURRENCY IN ERLANG

lightweight processes with asynchronous message passing

Procesele in Erlang:

- pot fi create si distruse rapid
- comunica prin mesaje, iar comunicarea este rapida
- sunt complet independente din punctul de vedere al memoriei

➢ Crearea proceselor: spawn

Functia spawn creaza un process care este executat in parallel cu procesul care l-a creat si intoarce un **Pid** (Process Identifier) , care este folosit pentru trimiterea mesajelor.

spawn/3
spawn(modul, functie, lista argumentelor)
Pid = spawn(modul, functie, lista argumentelor)

```
31> c(myconc).
{ok,myconc}
32> spawn(myconc,prelA,[5]).
A
<0.123.0>
A
A
A
A
End A
```

Pid= spawn(myconc,prelA,[5]).

```
-module(myconc).
-export([prelA/1).

prelA(X) when (X == 0) -> io:format("End A ~n");
prelA(X) when (X > 0) -> io:format("A ~n"), prelA(X-1);
prelA(_) ->       io:format("error ~n").
```

Exemplu: doua procese care sunt executate in paralel

```
33> [spawn(myconc,prelA,[10]),spawn(myconc,prelB,[10])].
A
B
[<0.125.0>,<0.126.0>]
A
B
A
B
A
B
A
B
A
B
A
B
A
B
A
B
A
B
A
B
End A
End B
```

- interleaving
- executie paralela

In Erlang, shell-ul este un process.
In interiorul unui process, functia self()
se refera la procesul respectiv.

```
Eshell V7.3  (abort with ^G)
1> self().
<0.32.0>
2> self()! hi.
hi
3> self()! good_bye.
good_bye
4> flush().
Shell got hi
Shell got good_bye
ok
```

flush()
elimina mesajele trimise shell-ului

```
6> Pid = self().
<0.56.0>
7> Pid ! hi.
hi
```

Un proces este identificat printr-un "process identifier (pid)"; un pid este un tip de date in Erlang; functia self() intoarce pid-ul procesului.

```
Eshell V7.3  (abort with ^G)
1> G=fun(X)->io:format("~p~n",[X]) end.
#Fun<erl_eval.6.50752066>
2> G(3).
3
ok
3> spawn(fun()->G(3) end).
3
<0.36.0>
4> Gt=fun(X)->timer:sleep(10), io:format("~p~n",[X]) end.
#Fun<erl_eval.6.50752066>
5> Gt(3).
3
ok
6> L=lists:seq(1,10).
[1,2,3,4,5,6,7,8,9,10]
7> [spawn(fun()->Gt(X) end)||X<-L].
[<0.41.0>,<0.42.0>,<0.43.0>,<0.44.0>,<0.45.0>,<0.46.0>,
 <0.47.0>,<0.48.0>,<0.49.0>,<0.50.0>]
1
2
3
4
5
6
7
8
9
10
```

timer:sleep(10)
suspenda procesul pentru
10 milisecunde
http://erlang.org/doc/man/timer.html

spawn/1
spawn (fun() -> Gt(X) end)

! Argumentul lui spawn este o functie, nu un apel de functie.

➢Trimiterea mesajelor: **Pid ! msg**

Mesajul **msg** este trimis procesului cu id-ul **Pid.** Mesajul este un termen Erlang.

```
myrec() ->
  receive
   {do_A, X} ->  prelA(X);
   {do_B, X} ->  prelB(X);
              _ -> io:format("Nothing to do ~n")
  end.
```

```
2> c(myconc).
{ok,myconc}
3> Rec=spawn(myconc, myrec,[]).
<0.40.0>
4> Rec! {do_A,2}.
A
{do_A,2}
A
End A
5> Rec! {do_B,2}.
{do_B,2}
6> f(Rec).
ok
7> Rec=spawn(myconc, myrec,[]).
<0.45.0>
8> Rec! {do_B,2}.
B
{do_B,2}
B
End B
```

```
9> f(Rec).
ok
10> Rec=spawn(myconc, myrec,[]).
<0.49.0>
11> Rec! fjrhjh.
Nothing to do
fjrhjh
```

f(X)
elibereaza X
http://erlang.org/doc/man/shell.html

Schimb de mesaje

```
myreceiver() ->
  receive
  {From, {do_A, X}} -> From ! "Thanks! I do A!",
                       prelA(X);
  {From, {do_B, X}} -> From! "Thanks! I do B!",
                       prelB(X);
       _    -> io:format("Nothing to do ~n")
  end.
```

schimb de mesaje intre
Rec si shell
flush() listeaza mesajele
primite de shell

```
12> RecM=spawn(myconc, myreceiver,[]).
<0.52.0>
13> RecM ! {self(),{do_A,4}}.
A
{<0.32.0>,{do_A,4}}
A
A
A
End A
14> flush().
Shell got "Thanks! I do A!"
ok
```

➢ receive … end

```
receive
Pattern1 when Guard1 -> Expr1;
Pattern2 when Guard2 -> Expr2;
Pattern3 -> Expr3
end
```

Cand ajunge la o instructiune receive un proces scoate un mesaj din mailbox si incearca sa ii gaseasca un pattern.

Daca nu gaseste un mesaj in mailbox procesul se blocheaza si asteapta un mesaj care se potriveste cu un pattern.

- trimiterea mesajelor se face asincron
- receive este singura instructiune care blocheaza procesul

"Messages between Erlang processes are simply valid Erlang terms. That is, they can be lists, tuples, integers, atoms, pids, and so on.

Each process has its own input queue for messages it receives. New messages received are put at the end of the queue. When a process executes a receive, the first message in the queue is matched against the first pattern in the receive. If this matches, the message is removed from the queue and the actions corresponding to the pattern are executed.

However, if the first pattern does not match, the second pattern is tested. If this matches, the message is removed from the queue and the actions corresponding to the second pattern are executed. If the second pattern does not match, the third is tried and so on until there are no more patterns to test. If there are no more patterns to test, the first message is kept in the queue and the second message is tried instead. If this matches any pattern, the appropriate actions are executed and the second message is removed from the queue (keeping the first message and any other messages in the queue). If the second message does not match, the third message is tried, and so on, until the end of the queue is reached. If the end of the queue is reached, the process blocks (stops execution) and waits until a new message is received and this procedure is repeated."

http://erlang.org/doc/getting_started/conc_prog.html

➢ Concurenta in Erlang este implementata folosind urmatoarele primitive:

```
Pid = spawn (fun)

Pid = spawn (module, fct, args)

Pid ! Message

receive … end
```

➢ Ping - Pong

ppmod.erl

```
-module(ppmod).
-export([start/0,pingN/2,pong/0]).

pingN(Pid,0) -> Pid ! {self(), finished},
                io:format("Ping finished!~n");

pingN(Pid, N) -> Pid ! {self(),ping},
          receive
             {Pid, pong} -> io:format("Ping received Pong. ~n")
          end,
          pingN(Pid,N-1).
```

tail-recursion

```
pong() ->
   receive
      {_,finished} -> io:format("Game over. ~n");
     {Pid, ping} -> io:format("Pong received Ping. ~n"),
                    Pid ! {self(),pong},
                     pong()
   end.

start() -> PongId = spawn(ppmod, pong,[]),
           spawn(ppmod,pingN,[PongId,5]).
```

tail-recursion

http://erlang.org/doc/getting_started/conc_prog.html

```
2> c(ppmod).
{ok,ppmod}
3> ppmod:start().
Pong received Ping.
<0.65.0>
Ping received Pong.
Pong received Ping.
Ping received Pong.
Pong received Ping.
Ping received Pong.
Pong received Ping.
Ping received Pong.
Pong received Ping.
Ping received Pong.
Ping finished!
Game over.
```

> erl –s ppmod start

```
C:\Users\Ioana\Documents\DIR>erl -s ppmod start
Pong received Ping.
Ping received Pong.
Pong received Ping.
Ping received Pong.
Pong received Ping.
Ping received Pong.
Pong received Ping.
Ping received Pong.
Pong received Ping.
Ping received Pong.
Ping finished!
Game over.
Eshell V8.3  (abort with ^G)
```

➢ Cilent-Server (Exemplu simplu: doubling service)

```
-module(myserv).
-export([server_loop/0]).

server_loop() ->
  receive
       {From, {double, Number}} -> From ! {self(), Number*2},
                                   server_loop() ;

        {From,_} -> From ! {self(),error},
                    server_loop()
  end.
```

```
3> c(myserv).
{ok,myserv}
4> Ser=spawn(myserv, server_loop, []).
<0.44.0>
5> Ser ! {self(),{double,5}}.
{<0.32.0>,{double,5}}
6> flush().
Shell got {<0.44.0>,10}
ok
7> Ser ! {self(),{double,7}}.
{<0.32.0>,{double,7}}
8> flush().
Shell got {<0.44.0>,14}
ok
9> Ser ! {self(),111}.
{<0.32.0>,111}
10> flush().
Shell got {<0.44.0>,error}
ok
```

```
-export([start_server/0, server_loop/0]).
start_server() ->  spawn(myserv, server_loop, []).
```

```
16> Ser=myserv:start_server().
<0.66.0>
17> Ser ! {self(), {double,45}}.
{<0.59.0>,{double,45}}
18> flush().
Shell got {<0.66.0>,90}
ok
```

➢ Cilent-Server (Exemplu simplu: doubling service)

```
-module(myserv).
-export([start_server/0, server_loop/0,client/2]).


start_server() ->  spawn(myserv, server_loop, []).


server_loop() ->
  receive
     {From, {double, Number}} -> From ! {self(),(Number*2)},
                                 server_loop() ;

     {From,_} -> From ! {self(),error},
                 server_loop()
  end.
```

```
client(Pid, Request) ->
      Pid ! {self(), Request},
      receive
            {Pid, Response} -> Response
       end.
```

```
3> c(myserv).
{ok,myserv}
4> Server = myserv:start_server().
<0.43.0>
5> myserv:client(Server,{double,15675}).
31350
6> myserv:client(Server,nothing).
error
7> myserv:client(Server, {double, 887}).
1774
```

- Client-Server

  client_loop creaza mai multe procese client si intoarce lista rezultatelor

```
client_loop(Pid,0,L)  -> Pid! {self(),"Good Bye"},
                         L;

client_loop(Pid, X, L)  ->  R= client(Pid,{double,X}),
                            client_loop(Pid, X-1, L++[R]).
```

```
31> c(myserv2).
{ok,myserv2}
32> Ser = myserv2:start_server().
<0.113.0>
33> myserv2:client_loop(Ser,10,[]).
[20,18,16,14,12,10,8,6,4,2]
34> flush().
Shell got {<0.113.0>,"Good Bye"}
ok
```

procesele client se executa **secvential**

➢Client-Server

procesele client se executa in **parallel** si se intoarce lista rezultatelor

```
worker(Parent, Pid, Number) ->   spawn( fun() ->
                                          Result = client (Pid,{double,Number}),
                                          Parent ! {self(),Result}
                                          end ).
 calls (Pid,N) ->  Parent = self(),
                   Pids = [worker(Parent,Pid, X)|| X <- lists:seq(1,N)],
                   [ waitone(P) ||  P <- Pids ].

waitone (Pid) ->
                receive
                    {Pid,Response} -> Response
                 end.
```

➢ Client-Server

```
start_server() ->  spawn(myserv, server_loop, []).
start_seq_clients(Pid, N) -> client_loop(Pid,N,[]).
start_par_clients(Pid, N) -> calls(Pid,N).
```

```
62> c(myserv2).
{ok,myserv2}
63> Server = myserv2:start_server().
<0.15705.27>
64> myserv2:start_par_clients(Server, 100000).
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,
 44,46,48,50,52,54,56,58|...]
65> myserv2:start_seq_clients(Server, 100000).
```

➢Client-Server

unui proces i se poate asocia un nume (atom) folosind register

myserv3.erl

```
start_server() ->  register(serv, spawn(fun() ->server_loop() end)).

server_loop() ->
  receive
    {From, {double, Number}} -> From ! {serv,(Number*2)},
                                server_loop() ;
      {From, "Good Bye"} -> From! {serv,"Good Bye"},
                            server_loop();

    {From,_} -> From ! {serv,error},
                server_loop()
  end.
```

procesul server are numele **serv**

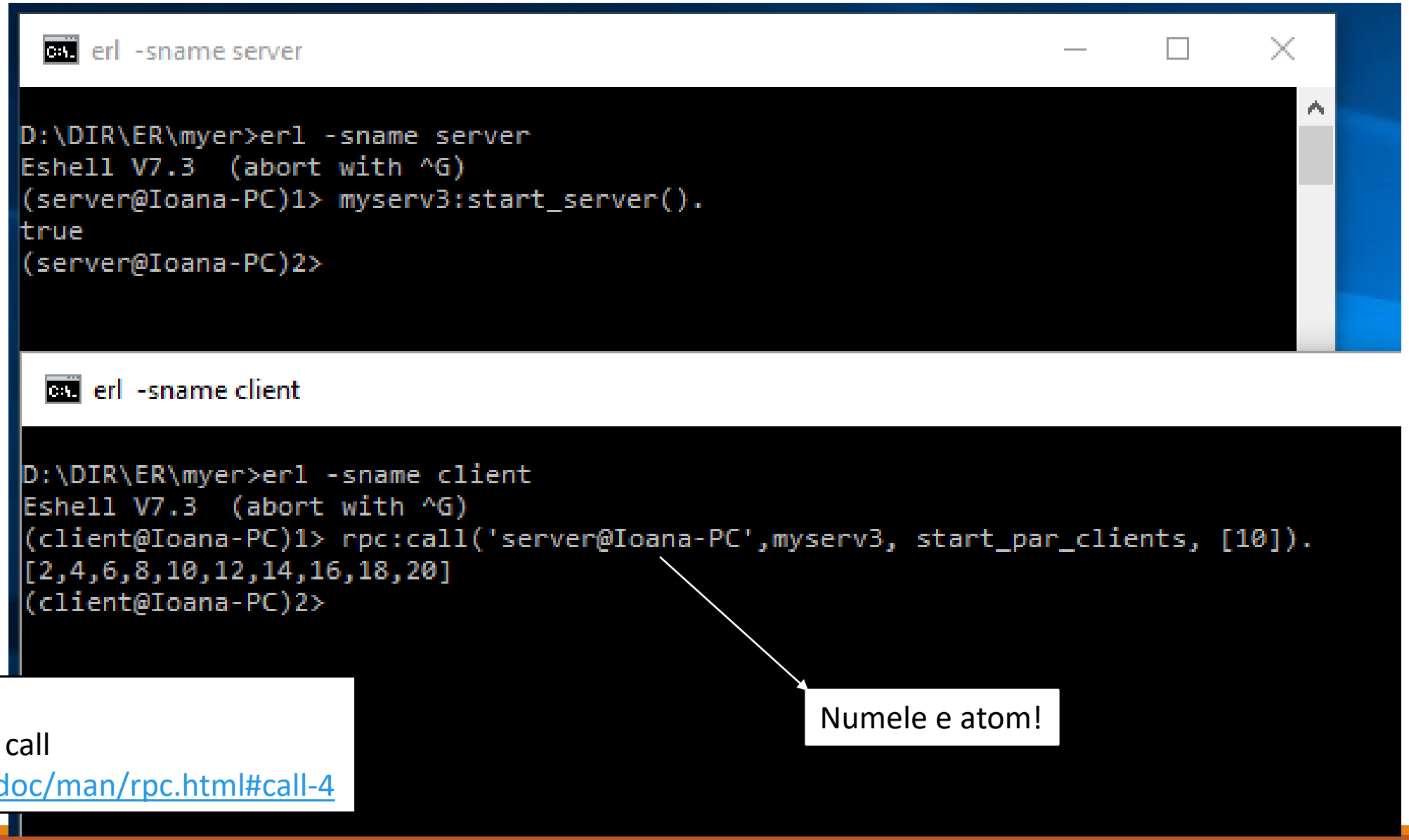
➢ Client-Server

```
start_par_clients(N) -> calls(N).
worker(Parent,  Number) ->
   spawn( fun() ->
                Result = client ({double,Number}),
                Parent ! {self(),Result}
              end ).

calls (N) ->
      Parent = self(),
      Pids = [worker(Parent,X)|| X <- lists:seq(1,N)],
      [waitone(P)|| P <- Pids].



waitone (Pid) ->
              receive
                 {Pid,Response} -> Response
              end.
```

```
client(Request) ->
        serv ! {self(), Request},
        receive
              {serv, Response} -> Response
        end.
```

```
1> cd ("D:/DIR/ER/myer").
D:/DIR/ER/myer
ok
2> c(myserv3).
{ok,myserv3}
3> myserv3:start_server().
true
4> myserv3:start_par_clients(50).
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,
 44,46,48,50,52,54,56,58|...]
5>
```

Distributed Erlang: programele ruleaza in noduri diferite



rpc:call
remote procedure call
http://erlang.org/doc/man/rpc.html#call-4