IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean

INTRODUCERE IN ERLANG

http://www.erlang.org/

- Bibliografie

Joe Armstrong, Programming Erlang, Second Edition 2013

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

PARALELISM

CONCURENTA

SISTEME DISTRIBUITE

"Erlang was designed from the bottom up to program concurrent, distributed, fault-tolerant, scalable, soft, real-time systems. […]

If your problem is concurrent, if you are building a multiuser system, or if you are building a system that evolves with time, then using Erlang might save you a lot of work, since Erlang was explicitly designed for building such systems. […]

Processes interact by one method, and one method only, by exchanging messages. Processes share no data with other processes. This is the reason why we can easily distribute Erlang programs over multicores or networks. "

Joe Armstrong, Programming Erlang, Second Edition 2013

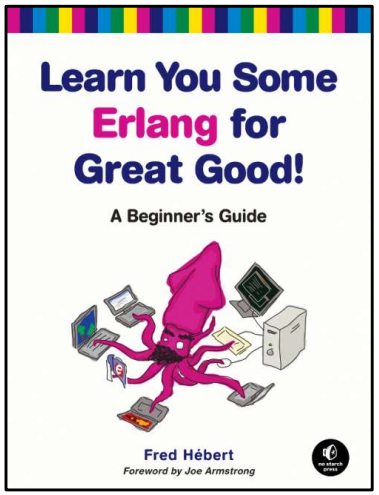
ACTOR MODEL

"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

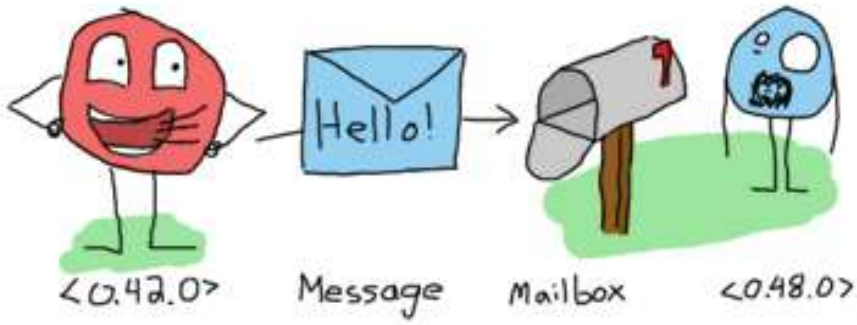
http://learnyousomeerlang.com/introduction#what-is-erlang



Varianta online

- Transmiterea mesajelor este asincrona.

Datorita cozii pentru mesaje,
procesul care transmite mesajul
nu asteapta o confirmare de primire sau
prelucrarea acestuia,
mesajul intra in coada si asteapta
pana cand va fi procesat



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

➢Modelul Actori

- Introdus de Carl Hewitt in 1973
- Actorii sunt o notiune abstracta (corespunzatoare proceselor)
- Actorii au memorie proprie, NU au memorie partajata
- Actorii comunica prin mesaje
- Un actor este capabil sa:
    - trimita mesaje actorilor pe care ii cunoaste
    - creeze noi actori
    - raspunda mesajelor pe care le primeste
- Mesajele contin un destinatar si un continut
- Trimiterea mesajelor este asincrona

- Concurenta in Erlang este implementata folosind urmatoarele primitive:

```
Pid = spawn (fun)

Pid = spawn (module, fct, args)

Pid ! Message

receive … end
```

https://www.erlang.org/doc/man/erlang.html#spawn-4

➢ Cilent-Server (Exemplu simplu: doubling service)

```
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
```

- Procesul **Ser** este serverul si executa functia **server_loop**
- Serverul primeste mesaje de la procese client si executa o actiune (dubleaza valoarea primita)
- In acest exemplu singurul client este shell-ul
- Mesajele primite de shell, adica raspunsurile trimise de server, sunt vizualizate cu **flush()**

➢ 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
```

➢ Cilent-Server : functie pentru pornirea server-ului

```
-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: functia client

```
-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.
```

functia **client intoarce raspunsul primit de la server**

apelarea functiei client

```
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
```

- 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]).
```

procesele client se executa **secvential**

- Client-Server

  client_loop creaza mai multi clienti si intoarce lista rezultatelor

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

client_loop(Pid, X, L)  ->  R= client(Pid,{double,X}),
                            io:fwrite("prel ~w!~n", [N]),
                            client_loop(Pid, X-1, L++[R]).
```

Functiile client sunt executate **secvential!**

➢ Client-Server

client_loop creaza mai multi clienti si intoarce lista rezultatelor

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

client_loop(Pid, X, L)  ->  R= client(Pid,{double,X}),
                            io:fwrite("prel ~w!~n", [N]),

                            client_loop(Pid, X-1, L++[R]).
```

```
28> Ser6=myserv2:start_server().
<0.139.0>
29> [myserv2:client_loop(Ser6, 3, []),myserv2:client_loop(Ser6, 3, [])].
prel 3!
prel 2!
prel 1!
prel 3!
prel 2!
prel 1!
```

Functiile client sunt executate **secvential!**

➢ Client-Server

client_loop creaza mai multi clienti si intoarce lista rezultatelor

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

client_loop(Pid, X, L)  ->  R= client(Pid,{double,X}),
                            io:fwrite("prel ~w!~n", [N]),

                            client_loop(Pid, X-1, L++[R]).
```

```
24> Ser5=myserv2:start_server().
<0.126.0>
25> [spawn(myserv2, client_loop, [Ser5, 3, []]),spawn(myserv2,client_loop,[Ser5, 3, []])].
prel 3!
prel 3!
[<0.128.0>,<0.129.0>]
prel 2!
prel 2!
prel 1!
prel 1!
```

Functiile client sunt executate **in paralel!**

➢Client-Server

procesele client se executa in **paralel** 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)],
                        [ wait_one(P)  ||  P <- Pids ].

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

```
% Pid este id-ul procesului server
% Parent este id-ul procesului care creaza clientii
% worker creaza un proces client si intoarce id-ul acestuia
```

- 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>
```

➢ ?MODULE

macro care intoarce numele modulului curent

```
start() -> spawn(?MODULE, myrec, []).

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

```
3> Pid = myconc1:start().
<0.112.0>
4> Pid ! {do_A,2}.
A
{do_A,2}
A
End A
```

➢ Cilent-Server (simple) template

```
-module(servtemplate1).
-compile(export_all).  %exporta toate functiile

start_server() ->  spawn(?MODULE, server_loop, []).

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

server_loop() ->
   receive
        .....
        {From, Request} -> From ! {self(),Response} ,
                                   server_loop()

        end.
```

```
-module(servtemplate2).
-compile(export_all).

start_server() ->
  register(serv,spawn(?MODULE, server_loop, [])).

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

server_loop() ->
   receive
        .....
        {From, Request} -> From ! {serv, Response} ,
                                   server_loop()

        end.
```

➢ Schimb de mesaje cu transmiterea starii (message passing with data storage)

- Procesul (serverul) este un frigider care accepta doua tipuri de comenzi
  - depoziteaza alimente,
  - scoate alimente .

- Acelasi aliment poate fi depozitat de mai multe ori si poate fi scos de cate ori a fost depozitat.

- La fiecare moment trebuie sa stim ce alimente se gasesc in frigider (starea procesului).

- Starea procesului se transmite prin parametrii functiilor.

kitchen.erl
http://learnyousomeerlang.com/

- Schimb de mesaje cu transmiterea starii

```
fridgef(FoodList) ->
                receive
                % comanda store
                % comanda take
                ....
                end.
```

```
store(Pid, Food) ->
        Pid ! {self(), {store, Food}},
        receive
                {Pid, Msg} -> Msg
        end.
```

```
take(Pid, Food) ->
        Pid ! {self(), {take, Food}},
        receive
                {Pid, Msg} -> Msg
        end.
```

kitchen.erl

http://learnyousomeerlang.com/

➢ Mesaje cu transmiterea starii

kitchen.erl
http://learnyousomeerlang.com/

```
fridgef(FoodList) ->
        receive
            {From, {store, Food}}  ->   From ! {self(), ok},
                                        fridgef([Food|FoodList]);

            {From, {take, Food}} ->  case lists:member(Food, FoodList) of
                                        true ->  From ! {self(), {ok, Food}},
                                                 fridgef(lists:delete(Food, FoodList));
                                        false ->   From ! {self(), not_found},
                                                   fridgef(FoodList)
                                     end;
             terminate   ->      ok
        end.
```

```
6> c(kitchen).
{ok,kitchen}
7> Fridge = kitchen:start([milk, cheese, ham]).
<0.99.0>
8> kitchen:store(Fridge, juice).
ok
9> kitchen:take(Fridge, milk).
{ok,milk}
10> kitchen:take(Fridge, juice).
{ok,juice}
11> kitchen:take(Fridge, juice).
not_found
```

- Varianta: registered process, comenzile **show** (pentru a vizualiza starea) si **terminate**

```
start(FoodList) -> register(fridge, spawn(fun()-> fridgef(FoodList) end)).
```

```
store(Food) ->
   fridge! {self(), {store, Food}},
   receive
      {fridge, Msg} -> Msg
   end.

take( Food) ->
   fridge ! {self(), {take, Food}},
   receive
      {fridge, Msg} -> Msg
   end.
```

```
show() ->
   fridge ! {self(), show},
   receive
      {fridge, Msg} -> Msg
   end.

terminate() ->
      fridge ! {self(), terminate},
      receive
        {fridge, Msg} -> Msg
      end.
```

- Varianta: registered process, comenzile **show** (pentru a vizualiza starea) si **terminate**

```
fridgef(FoodList) ->
   receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                  case lists:member(Food, FoodList) of
                     true ->   From ! {fridge, {ok, Food}},
                               fridgef(lists:delete(Food, FoodList));
                     false ->  From ! {fridge, not_found},
                               fridgef(FoodList)
                   end;
      {From,show} ->  From ! {fridge, FoodList},
                      fridgef(FoodList);
      {From,terminate} -> From ! {fridge, done}
    end.
```

mykitchen.erl

```
2> c(mykitchen).
{ok,mykitchen}
3> mykitchen:start([milk, apple]).
true
4> mykitchen:take(milk).
{ok,milk}
5> mykitchen:store(orange).
ok
6> mykitchen:show().
[orange,apple]
7> mykitchen:terminate().
done
```

➢ receive … after … end

```
receive
Pattern1 when Guard1 -> Expr1;
Pattern2  when Guard2 -> Expr2;
Pattern3 -> Expr3
…
after T ->
      ExpressionT
end
```

- procesul asteapta pana cand primeste un mesaj care se potriveste cu un pattern sau pana cand expira timpul .
- timpul T este exprimat in milisecunde
- procesul va astepta maxim T milisecunde sa primeasca un mesaj
- daca nici un mesaj care se potriveste cu un patern nu este primit in timpul T, procesul executa ExpressionT

➢receive … after … end

```
sleep(T) ->
      receive
        after T ->
            ok
      end.
```

- nu exista sabloane, deci niciun mesaj nu va fi acceptat;
- procesul va fi blocat T milisecunde

```
flush() ->
  receive
          _      -> flush()
    after 0      ->
          ok
end.
```

- orice mesaj se potriveste cu patternul, deci apelul recursiv va goli coada de mesaje, dupa care procesul va continua

- instructiunea `after 0 -> …`

  verifica coada de mesaje si apoi continua;
  daca aceasta clauza lipseste, procesul se va bloca cand coada de mesaje se goleste

mykitchen3.erl

```
fridgef(FoodList) ->
 receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                  case lists:member(Food, FoodList) of
                    true ->   From ! {fridge, {ok, Food}},
                              fridgef(lists:delete(Food, FoodList));
                    false ->  From ! {fridge, not_found},
                              fridgef(FoodList)
                  end;
      {From,show} ->  From ! {fridge, FoodList},
                      fridgef(FoodList)
 end,
 io:format("al doilea receive~n"),
 receive
      {From,terminate} -> From ! {fridge, done}
  end.
```

```
1> c(mykitchen3).
{ok,mykitchen3}
2> mykitchen3:start([]).
true
3> mykitchen3:store(apple).
ok
4> mykitchen3:terminate().
```

procesul este **blocat pentru ca nu poate iesi din primul receive**

Ctrl-G deblocheaza shell-ul
f() dezleaga variabilele

```
User switch command
 --> i
 --> s
 --> c
Eshell V8.3  (abort with ^G)
1> f().
ok
```

http://erlang/doc/man/shell.html

➢ receive … after … end

```
fridgef(FoodList) ->
 receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                  case lists:member(Food, FoodList) of
                     true ->   From ! {fridge, {ok, Food}},
                               fridgef(lists:delete(Food, FoodList));
                     false ->  From ! {fridge, not_found},
                               fridgef(FoodList)
                   end;
      {From,show} ->  From ! {fridge, FoodList},
                      fridgef(FoodList)

 after 30000 ->   timeout
 end,
  io:format("al doilea receive~n"),
  receive
      {From,terminate} -> From ! {fridge, done}
   end.
```

```
1> c(mykitchen3).
{ok,mykitchen3}
2> mykitchen3:start([]).
true
3> mykitchen3:terminate().
al doilea receive
done
```

dupa 30 sec

- in apelul **terminate()**, procesul shell asteapta mesaj de la fridge pentru **receive** din **terminate()**

```
terminate() ->
      fridge ! {self(), terminate},
      receive
        {fridge, Msg} -> Msg
      end.
```

- apelul functiei **terminate()** se incheie numai dupa ce trec cele 30 sec si poate fi prelucrat mesajul "terminate" in al doilea **receive** din **fridgef()**

- receive … after … end

```
fridgef(FoodList) ->
 receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                  case lists:member(Food, FoodList) of
                    true ->   From ! {fridge, {ok, Food}},
                              fridgef(lists:delete(Food, FoodList));
                    false ->  From ! {fridge, not_found},
                              fridgef(FoodList)
                  end;
       {From,show} ->  From ! {fridge, FoodList},
                       fridgef(FoodList);
      {From,terminate} -> From ! {fridge, done}
  after 30000 ->   timeout
 end,
receive
   gata -> io:format("Sunt gata~n")
end.
```

```
gata() -> fridge ! gata
```

```
1> c(mykitchen4).
{ok,mykitchen4}
2> mykitchen4:start([apple]).
true
3> mykitchen4:gata().
gata
4> mykitchen4:show().
[apple]
5> mykitchen4:store(water).
ok
6> mykitchen4:show().
[water,apple]
Sunt gata
```

- functia gata() intoarce imediat, shell-ul nu ramane blocat
- se pot trimite mesaje serverului
- mesajul **gata** este prelucrat dupa ce au trec 30 sec fara o comanda prelucrata de primul receive

➢ receive … after … end

```
receive
Pattern1 when Guard1 -> Expr1;
Pattern2  when Guard2 -> Expr2;
Pattern3 -> Expr3
…
after T ->
      ExpressionT
end
```

- La intrarea in **receive**, daca exista un **after**, se porneste un timer.
- Mesajele din coada sunt investigate in ordinea sosirii; daca un mesaj se potriveste cu un pattern atunci expresia corespunzatoare este prelucrata.
- Mesajele care nu se potrivesc cu nici un pattern sunt puse intr-o coada separate (*save queue*).
- Daca nu mai sunt mesaje in coada procesul se suspenda si asteapta venirea unui nou mesaj; la venirea acestuia, numai el este prelucrat, nu si mesajele din *save queue.*
- Cand un mesaj se potriveste cu un pattern, mesajele din *save queue* sunt puse la loc in coada si timerul se sterge.
- Daca timpul T s-a scurs fara ca un mesaj sa se potriveasca unui pattern, atunci ExpressionT se executa, iar mesajele din *save queue* sunt puse inapoi in coada.

- Selective receives

```
5> self()! hi, self() ! low.
low
6> flush().
Shell got hi
Shell got low
ok
```

```
flush() ->
  receive
        _      -> flush()
    after 0    ->
        ok
end.
```

```
important() ->
    receive
      {Priority, X} when Priority > 10 ->   [X|important()]
     after 0 ->
            normal()
    end.

normal() ->
      receive
          {_,X} ->
                [X|normal()]
       after 0 -> []
end.
```

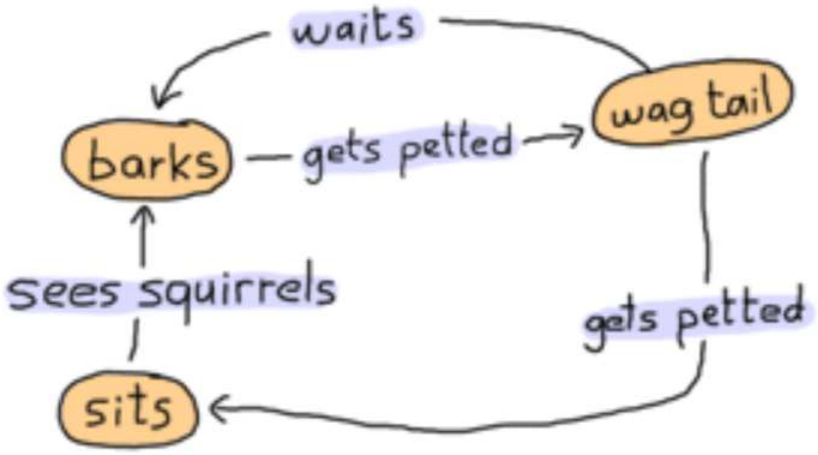
Varianta a functiei flush() care ordoneaza mesajele dupa prioritati

```
2> c(sel).
{ok,sel}
3> self()! {5, low1}, self() ! {9,low2}, self() ! {15, high1}, self()!{11,high2}
.
{11,high2}
4> sel:important().
[high1,high2,low1,low2]
```

- Concurenta in Erlang este implementata folosind urmatoarele primitive:

```
Pid = spawn (fun)

Pid = spawn (module, fct, args)

Pid ! Message

receive … end
receive … after … end
```

- Implementarea unui automat finit (Finite-State Machine)

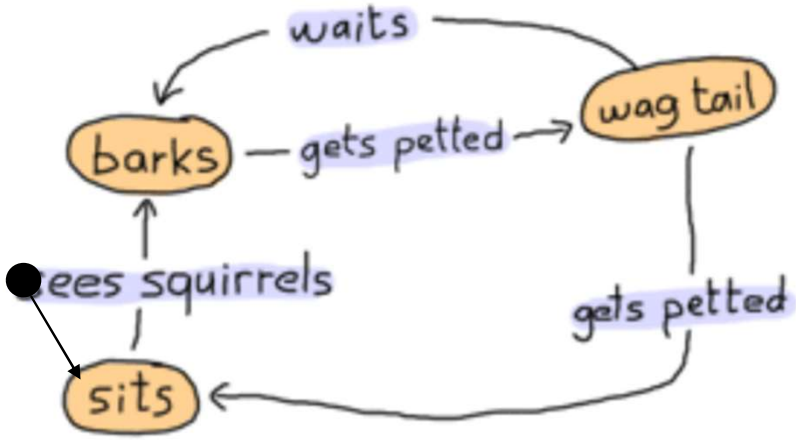


Starile ={barks, sits, wag_tail}
Actiunile ={gets_petted, see_squirrels, waits}

dog as a state-machine
http://learnyousomeerlang.com/finite-state-machines#what-are-they

➢ Implementarea unui automat finit

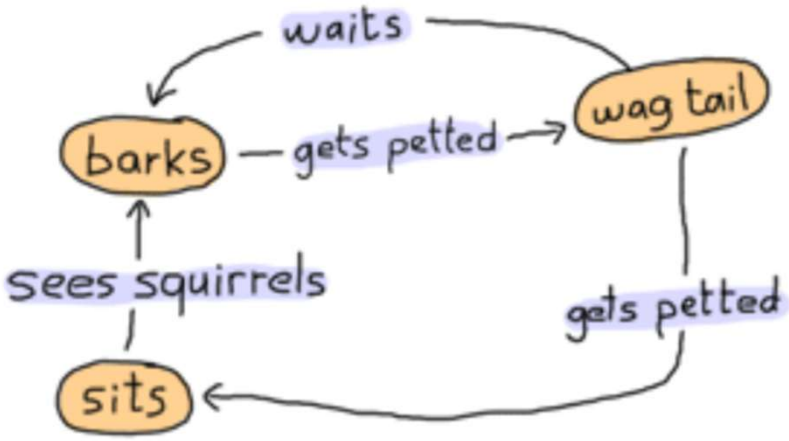
```
-module(dog_fsm).
-export([start/0, squirrel/1, pet/1]).

start() ->
    spawn(fun() -> bark() end).    % starea initiala

%actiunea  see_squirrels
squirrel(Pid) -> Pid ! squirrel.

%actiunea  gets_petted
pet(Pid) -> Pid ! pet.
```



actiunile sunt implementate prin mesaje si sunt vizibile in exterior

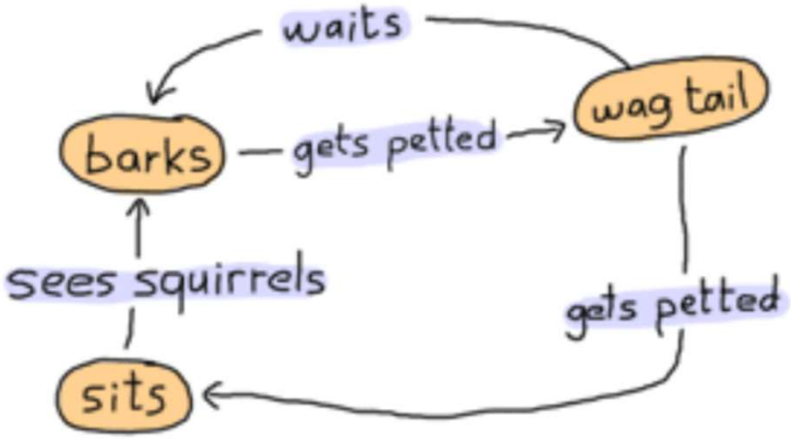
➢Finite-StateMachine: implementarea starilor

```
bark() ->
    io:format("Dog says: BARK! BARK!~n"),
   receive
     pet ->  wag_tail();

       _ ->  io:format("Dog is confused~n"),
             bark()

  after 2000 ->  bark()
end.
```



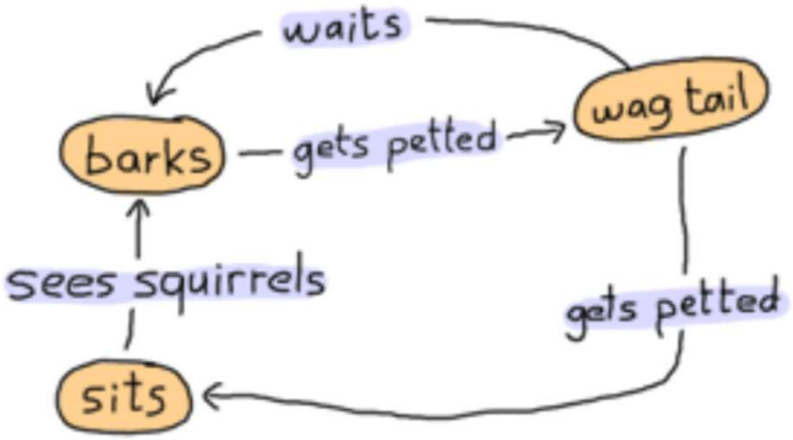
➢ Automat finit: definirea starilor

```
wag_tail() ->
        io:format("Dog wags its tail~n"),
        receive
             pet ->  sit();

             _ ->  io:format("Dog is confused~n"),
                   wag_tail()

        after 30000 ->
                      bark()      % actiunea waits

        end.
```



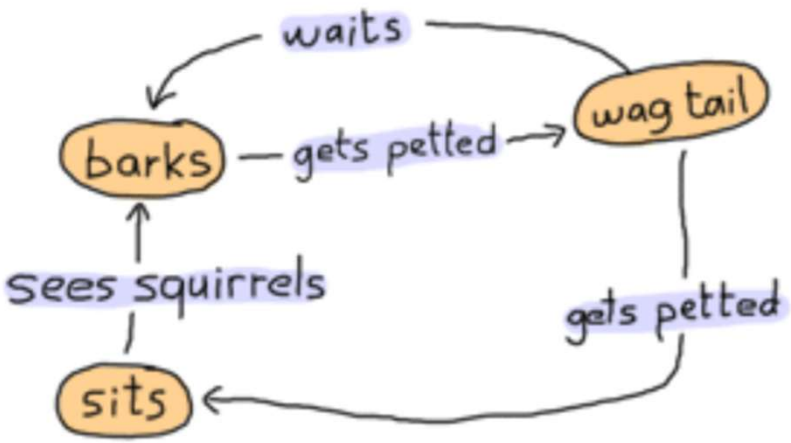
➢ Automat finit: definirea starilor

```
sit() ->
    io:format("Dog is sitting. Gooooood boy!~n"),
    receive
        squirrel -> bark();

        _ ->  io:format("Dog is confused~n"),
              sit()
    end.
```



➢ Implementarea unui automat finit

```
1> c(dog_fsm).
{ok,dog_fsm}
2> Pid=dog_fsm:start().
Dog says: BARK! BARK!
<0.63.0>
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
3> dog_fsm:pet(Pid).
Dog wags its tail
pet
4> dog_fsm:pet(Pid).
Dog is sitting. Gooooood boy!
pet
5> dog_fsm:squirrel(Pid).
Dog says: BARK! BARK!
squirrel
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
```



http://learnyousomeerlang.com/finite-state-machines#what-are-they

➢ Tratarea erorilor

Error handling in concurrent Erlang programs is based on the idea of *remote detection and handling of errors*. Instead of handling an error in the process where the error occurs, we let the process die and correct the error in some other process."
Joe Armstrong, Programming Erlang, Second Edition 2013

➢ OTP

OTP stands for Open Telecom Platform, although it's not that much about telecom anymore (it's more about software that has the property of telecom applications, but yeah.) If half of Erlang's greatness comes from its concurrency and distribution and the other half comes from its error handling capabilities, then the OTP framework is the third half of it.
http://learnyousomeerlang.com/what-is-otp#its-the-open-telecom-platform