Fault-tolerance, hot code loading (Erlang) (CPE 7*)

Carlos Varela
Rensselaer Polytechnic Institute

October 23, 2026

* Concurrent Programming in Erlang, by J. Armstrong, R. Virding, C. Wikström, M. Williams

Advanced Features of Actor Languages

- SALSA and Erlang support the basic primitives of the actor model:
  - Actors can create new actors.
  - Message passing is asynchronous.
  - State is encapsulated.
  - Run-time ensures fairness.
- SALSA also introduces advanced coordination abstractions: tokens, join blocks, and first-class continuations; SALSA supports distributed systems development including actor mobility and garbage collection. Research projects have also investigated load balancing, malleability (IOS), scalability (COS), and visualization (OverView).
- Erlang introduces a selective receive abstraction to enforce different orders of message delivery, including a timeout mechanism to bypass blocking behavior of `receive` primitive. Erlang also provides error handling abstractions at the language level, and dynamic (hot) code loading capabilities.

Erlang Language Support for Fault-Tolerant Computing

- Erlang provides linguistic abstractions for:
  - Error detection.
    - Catch/throw exception handling.
    - Normal/abnormal process termination.
    - Node monitoring and exit signals.
  - Process (actor) groups.
  - Dynamic (hot) code loading.

Exception Handling

- To protect sequential code from errors:

```
catch Expression
```

If failure does not occur in `Expression` evaluation, `catch Expression` returns the value of the expression.

- To enable non-local return from a function:

```
throw({ab_exception, user_exists})
```

Address Book Example

```
-module(addressbook).
-export([start/0,addressbook/1]).

start() ->
    register(addressbook, spawn(addressbook, addressbook, [[]])).

addressbook(Data) ->
  receive
    {From, {addUser, Name, Email}} ->
       From ! {addressbook, ok},
       addressbook(add(Name, Email, Data));
    …
end.

add(Name, Email, Data) ->
    case getemail(Name, Data) of
       undefined ->
         [{Name,Email}|Data];
       _ ->  % if Name already exists, add is ignored.
         Data
    end.
getemail(Name, Data) -> …
```

Address Book Example with Exception

```
addressbook(Data) ->
  receive
    {From, {addUser, Name, Email}} ->
       case catch add(Name, Email, Data) of
         {ab_exception, user_exists} ->
            From ! {addressbook, no},
            addressbook(Data);
         NewData->
            From ! {addressbook, ok},
            addressbook(NewData)
       end;
    …
end.

add(Name, Email, Data) ->
    case getemail(Name, Data) of
       undefined ->
         [{Name,Email}|Data];
       _ ->       % if Name already exists, exception is thrown.
         throw({ab_exception,user_exists})
    end.
```

Normal/abnormal termination

- To terminate an actor, you may simply return from the function the actor executes (without using tail-form recursion). This is equivalent to calling:

```
exit(normal).
```

- Abnormal termination of a function, can be programmed:

```
exit({ab_error, no_msg_handler})
```

equivalent to:

```
throw({'EXIT',{ab_error, no_msg_handler}})
```

- Or it can happen as a run-time error, where the Erlang run-time sends a signal equivalent to:

```
exit(badarg)            % Wrong argument type
exit(function_clause)   % No pattern match
```

Address Book Example with Exception and Error Handling

```
addressbook(Data) ->
  receive
    {From, {addUser, Name, Email}} ->
       case catch add(Name, Email, Data) of
         {ab_exception, user_exists} ->
            From ! {addressbook, no},
            addressbook(Data);
         {ab_error, What} -> …  % programmer-generated error (exit)
         {'EXIT', What} -> …    % run-time-generated error
         NewData->
            From ! {addressbook, ok},
            addressbook(NewData)
       end;
    …
end.
```

Node monitoring

- To monitor a node:

```
monitor_node(Node, Flag)
```

If `flag` is true, monitoring starts. If false, monitoring stops. When a monitored node fails, `{nodedown, Node}` is sent to monitoring process.

Address Book Client Example with Node Monitoring

```
-module(addressbook_client).
-export([getEmail/1,getName/1,addUser/2]).

addressbook_server() -> 'addressbook@127.0.0.1'.

getEmail(Name) -> call_addressbook({getEmail, Name}).
getName(Email) -> call_addressbook({getName, Email}).
addUser(Name, Email) -> call_addressbook({addUser, Name, Email}).

call_addressbook(Msg) ->
    AddressBookServer = addressbook_server(),
    monitor_node(AddressBookServer, true),
    {addressbook, AddressBookServer} ! {self(), Msg},
    receive
       {addressbook, Reply} ->
           monitor_node(AddressBookServer, false),
           Reply;
       {nodedown, AddressBookServer} ->
           no
    end.
```

Process (Actor) Groups

- To create an actor in a specified remote node:

```
Agent = spawn(host, travel, agent, []);
```

- To create an actor in a specified remote node and create a link to the actor:

```
Agent = spawn_link(host, travel, agent, []);
```

An `'EXIT'` signal will be sent to the originating actor if the host node does not exist.

Group Failure

- Default error handling for linked processes is as follows:
  - Normal exit signal is ignored.
  - Abnormal exit (either programmatic or system-generated):
    - Bypass all messages to the receiving process.
    - Kill the receiving process.
    - Propagate same error signal to links of killed process.
- All linked processes will get killed if a participating process exits abnormally.

Default Error Handling

- Default error handling for linked processes is as follows:
  - Normal exit signal is ignored.

```
exit(normal).
```

  - Abnormal exit (either programmatic or system-generated):
    - Bypass all messages to the receiving process.
    - Kill the receiving process.
    - Propagate same error signal to links of killed process.

```
{'EXIT', Exiting_PID, Reason}
```

Process exit signal propagation

```erlang
-module(normal).
-export([start/1, p1/1, test/1]).

start(N) ->
    register(start, spawn_link(normal, p1, [N - 1])).

p1(0) ->
    top1();
p1(N) ->
    top(spawn_link(normal, p1, [N - 1]),N).

top(Next, N) ->
    receive
        X ->
            Next ! X,
            io:format("Process ~w received ~w~n", [N,X]),
            top(Next,N)
    end.

top1() ->
    receive
        stop ->
            io:format("Last process now exiting ~n", []),
            exit(finished);
        X ->
            io:format("Last process received ~w~n", [X]),
            top1()
    end.

test(Mess) ->
    start ! Mess.
```

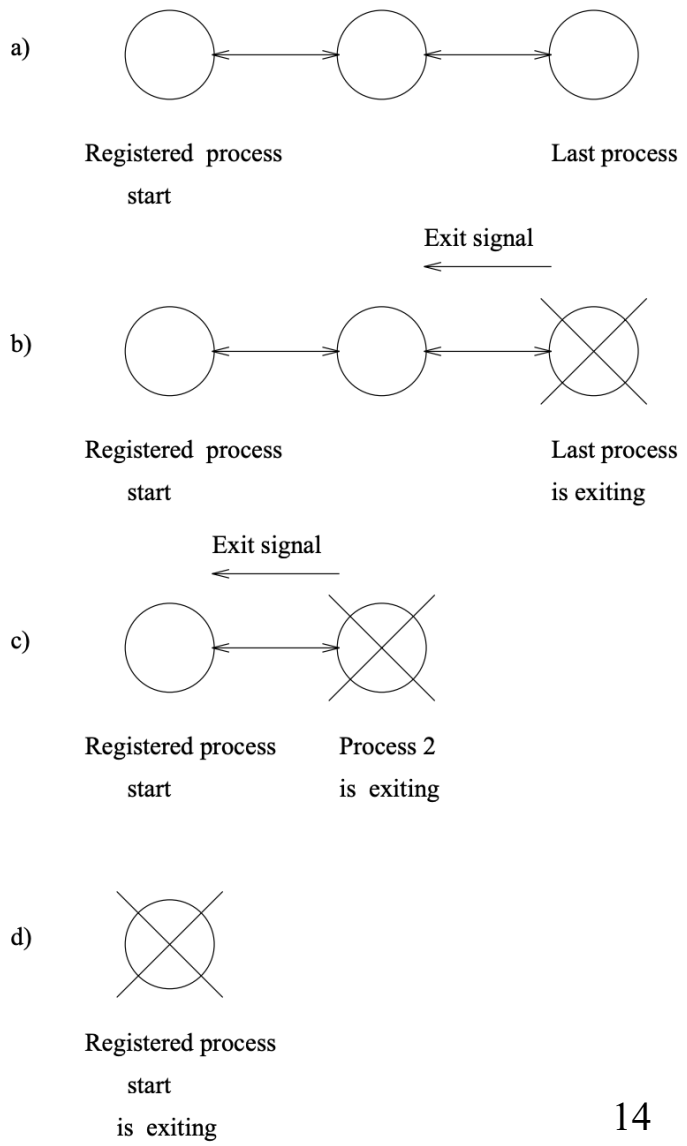


Figure 7.1 Process exit signal propagation

Process Linking / Unlinking

- To link to another process, other than using `spawn_link`:

```
link(PID);
```

- When a process terminates normally, all links are removed.
- To unlink explicitly:

```
un_link(PID);
```

Run-time Failures

- A run-time failure (not in the scope of a `catch` statement) causes the process to terminate abnormally. All linked processes receive an EXIT signal with an atom giving the reason for the failure, e.g.:
  - `badmatch`
    - A match has failed, eg, 1 = 3.
  - `badarg`
    - An argument to a BIF has incorrect type, eg. `atom_to_list(123)` since 123 is not an atom.
  - `case_clause`
    - No branch of a `case` expression matches.
  - `if_clause`
    - No branch of an `if` expression matches.

Run-time Failures (2)

- `function_clause`
  - No head of a function matches the arguments.
- `undef`
  - Function does not exist.
- `badarith`
  - Bad arithmetical expression, eg, `1 + foo`.
- `timeout_value`
  - A bad timeout value in a `receive` expression, ie., not an integer value or the atom `infinity`.
- `nocatch`
  - A `throw` statement does not have a corresponding `catch`.

Customized error handling

- To change the default group failure behavior:

```
process_flag(trap_exit, true)
```

Allows calling process to receive:

```
{'EXIT', Exiting_PID, Reason}
```

signals and take customized action.

Dynamic code loading

- To update (module) code while running it:

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

loop() ->
    receive
        code_switch ->
            m:loop();
        Msg -> ...
            loop()
    end.
```

`code_switch` message dynamically loads the new module code. Notice the difference between `m:loop()` and `loop()`.

Exercises

61. Create a ring of linked actors in Erlang.
    a. Cause one of the actors to terminate abnormally and observe default group failure behavior.
    b. Modify default error behavior so that upon an actor failure, the actor ring reconnects.
62. Modify the cell example, so that a new "get_and_set" operation is supported. Dynamically (as cell code is running) upgrade the cell module code to use your new version.