Concurrent Error Handling

Erlang SOLUTIONS

Process Error Handling

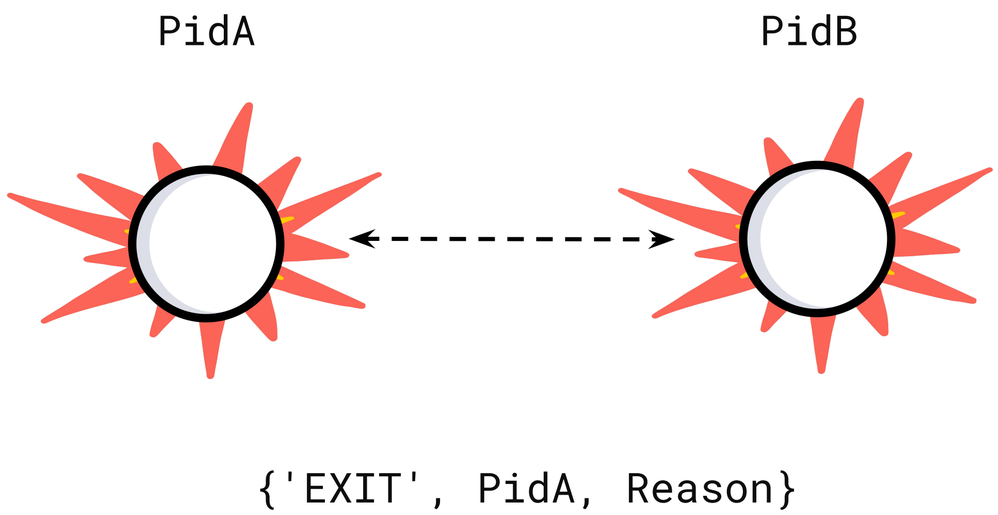
- Links
- Exit Signals
- Definitions
- Propagation Semantics
- Monitors
- Robust Systems
- Error Handling Example

Links

`link(PidB)`

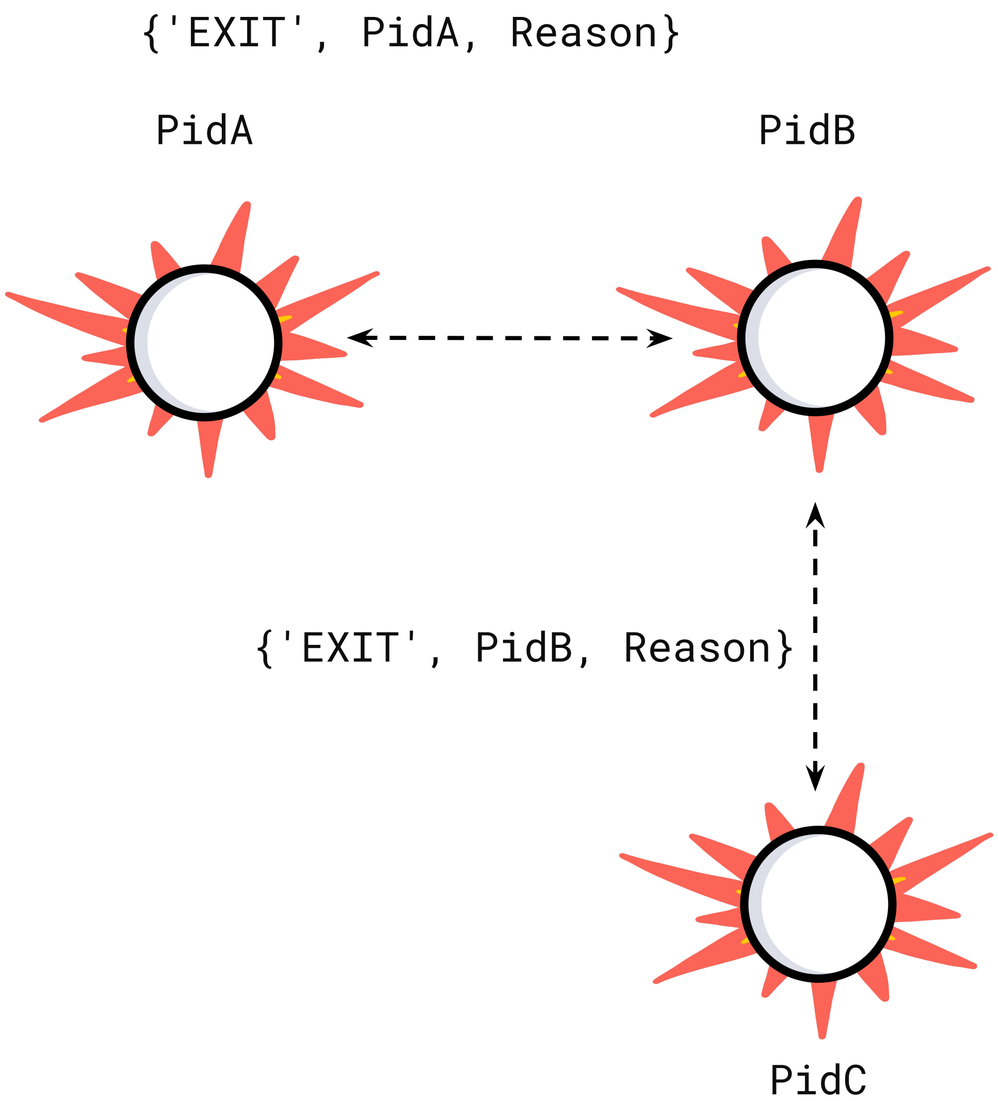
- **link/1** will create a bi-directional link between the process calling the BIF and the process **PidB**
- **spawn_link/3** will yield the same result as calling **spawn/3** followed by **link/1**, only that it will do it **atomically**

Exit Signals



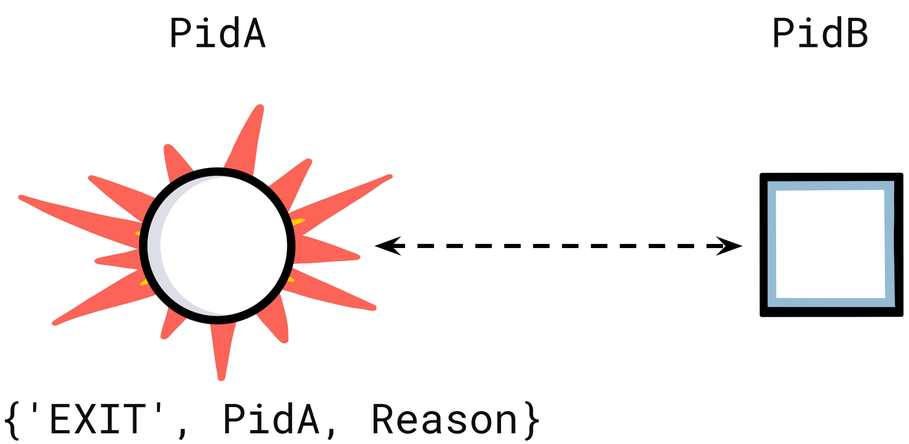
- **Exit Signals** are sent when processes terminate abnormally
- They are sent to all processes to which the failing process is currently linked to
- The process receiving the signal will exit, then propagate a new signal to the processes it is linked to

Exit Signals



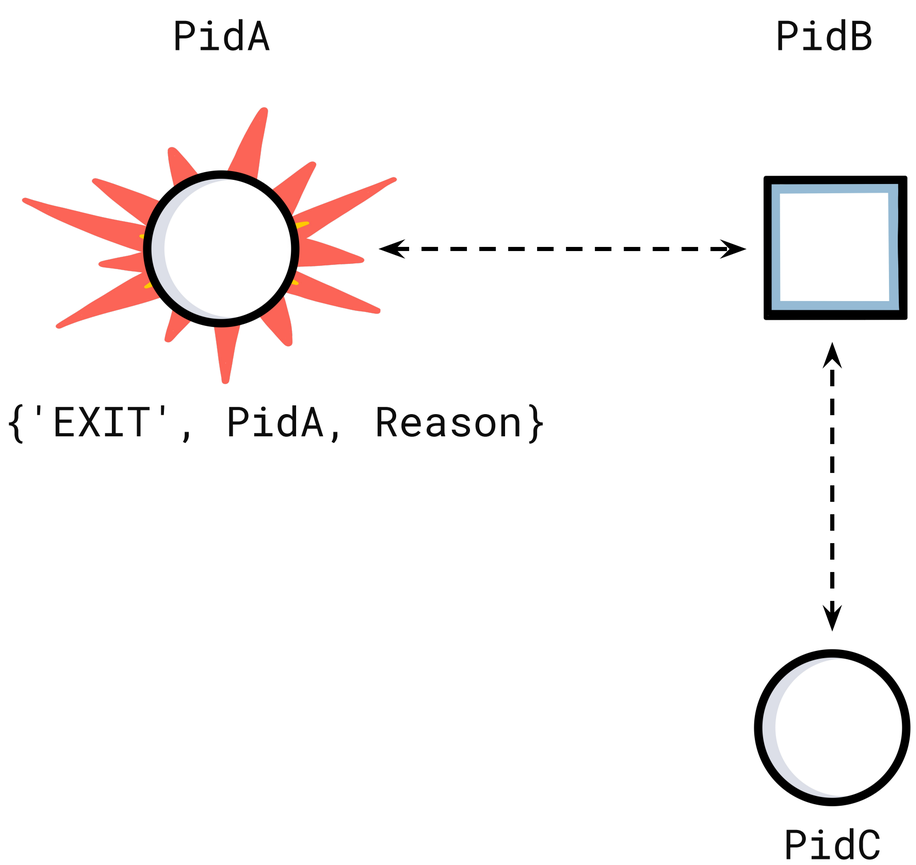
- When process **PidA** fails, the exit signals propagate to **PidB**
- From **PidB**, it propagates to **PidC**.

Exit Signals



- Processes can trap exit signals by calling the BIF **process_flag(trap_exit, true)**
- Exit signals will be converted to messages of the format **{'EXIT', Pid, Reason}**
- They are saved in the process mailbox
- If an exit signal is trapped, it does not propagate further

Exit Signals



- Process B marked with a double ring is trapping EXITs
- If an error occurs in A or C, then they will terminate.
- Process B will receive the **{'EXIT', Pid, Reason}** message
- The process that did not terminate will not be affected.

Definitions: terminology

Link

A bi-directional propagation path for exit signals set up between processes

Exit Signal

An asynchronous signal transmitted by a process upon exiting. It contains termination information

Error Trapping

The ability of a process to handle exit signals as if they were messages

Definitions: built-in functions

link(Pid)

Set a link between the calling process and Pid

unlink(Pid)

Removes a link to Pid

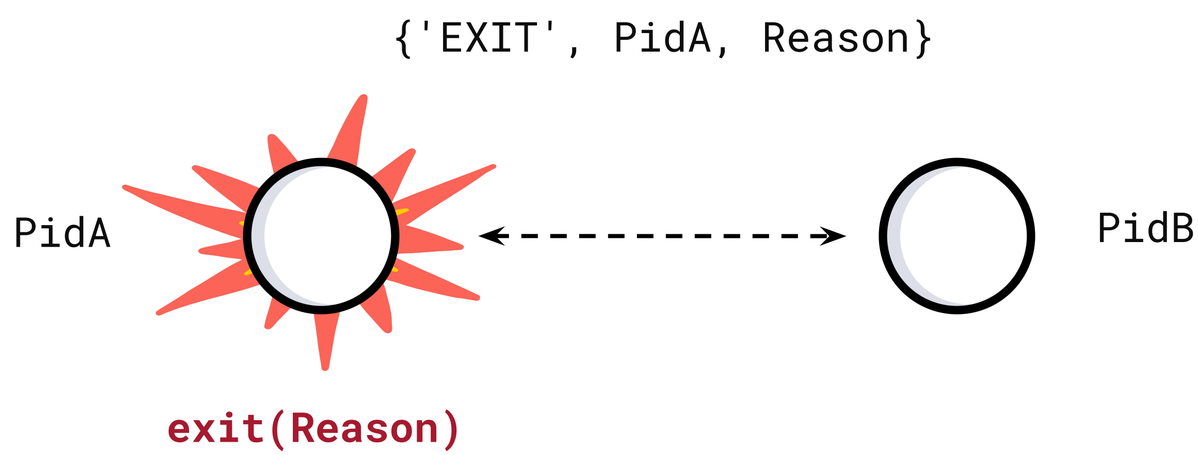
spawn_link(M, F, Args)

Atomically spawns and sets a link between the calling and the spawned processes.

process_flag(trap_exit, Bool)

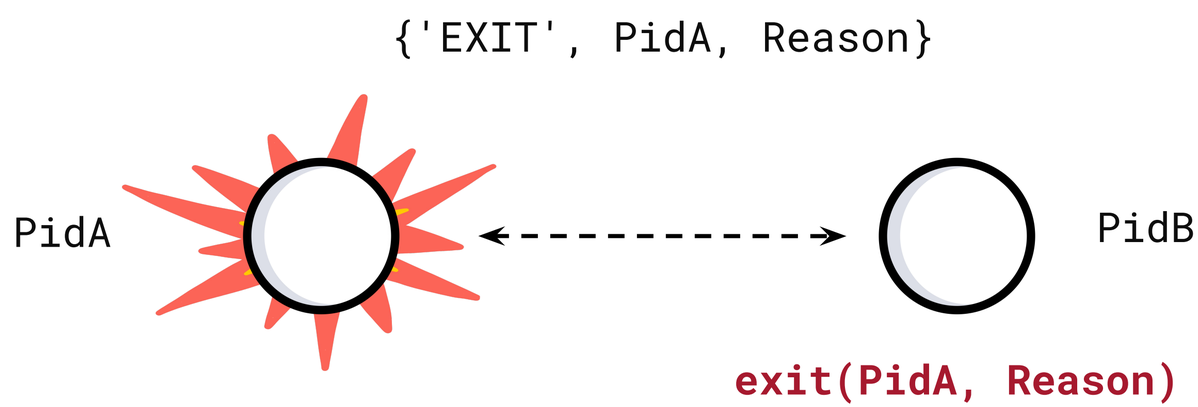
Sets the current process to convert exit signals into exit messages

Definitions: built-in functions



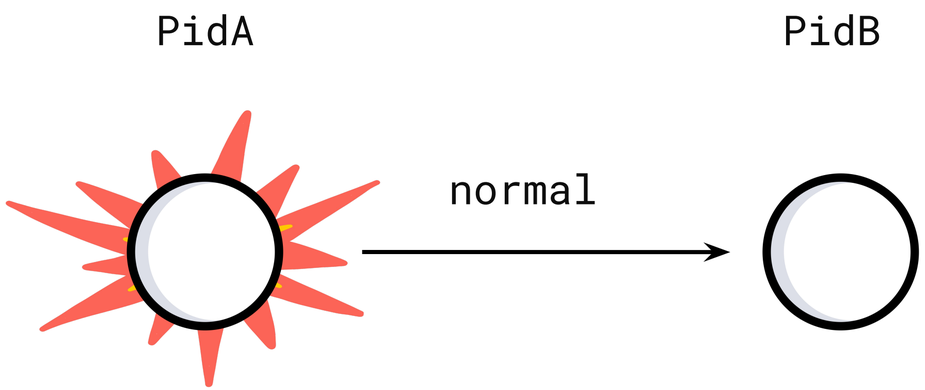
- The BIF **exit(Reason)** terminates the process which calls it
- It generates an exit signal sent to linked processes
- The BIF **exit/1** can be caught in a **try … catch**.

Definitions: built-in functions

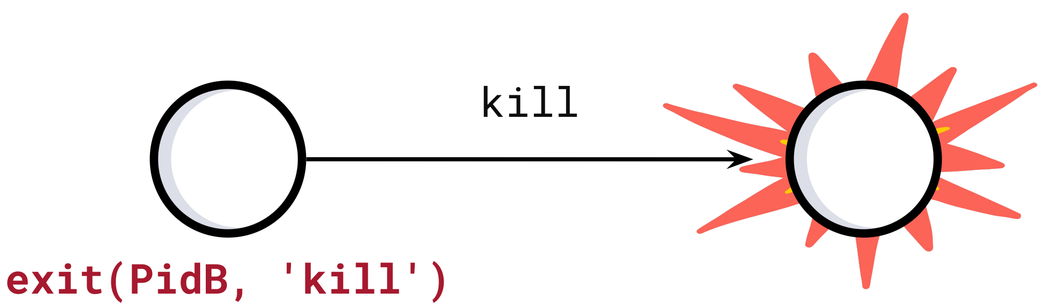


- **exit(Pid, Reason)** sends an exit signal containing **Reason** to the process **Pid**
- If PidA is trapping exits, the signal is converted to an exit message

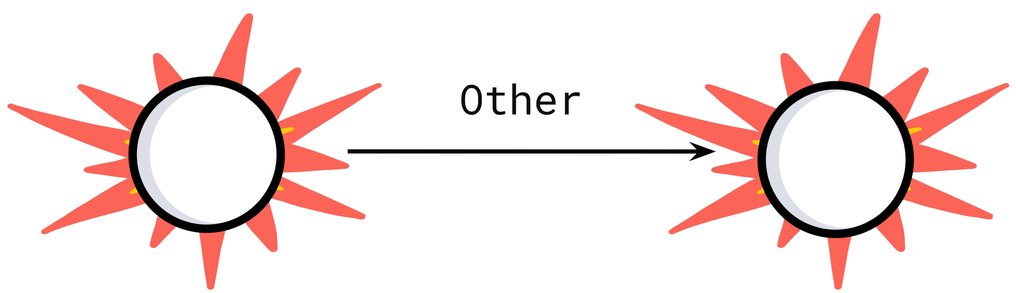
Propagation Semantics: no trapping



- Nothing happens to PidB

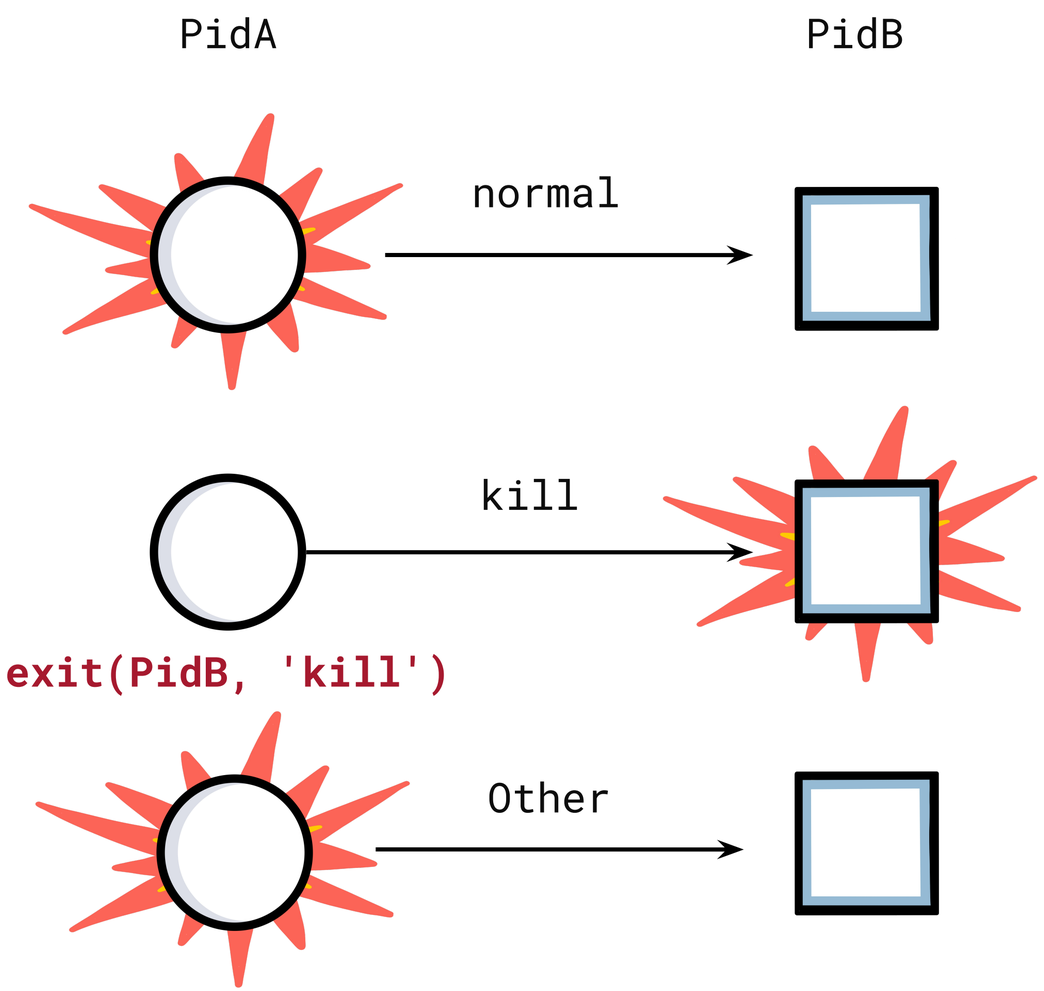


- PidB terminates with reason **killed**



- PidB terminates with reason **Other**

Propagation Semantics: trapping exits



- PidB receives **{'EXIT', PidA, normal}**
- PidB terminates with reason **killed**
- PidB receives **{'EXIT', PidA, Other}**

Propagation Semantics

- When a process terminates, it sends an exit signal to the processes in its link set
- Exit signals can be **normal** or ***non-normal***
- A process not trapping exits dies if it receives a non-normal one. Normal signals are ignored.
- A process which is trapping exit signals converts all incoming exit signals to conventional messages handled in a receive statement
- If the reason is **kill**, the process is terminated unconditionally with reason **killed**

Monitors

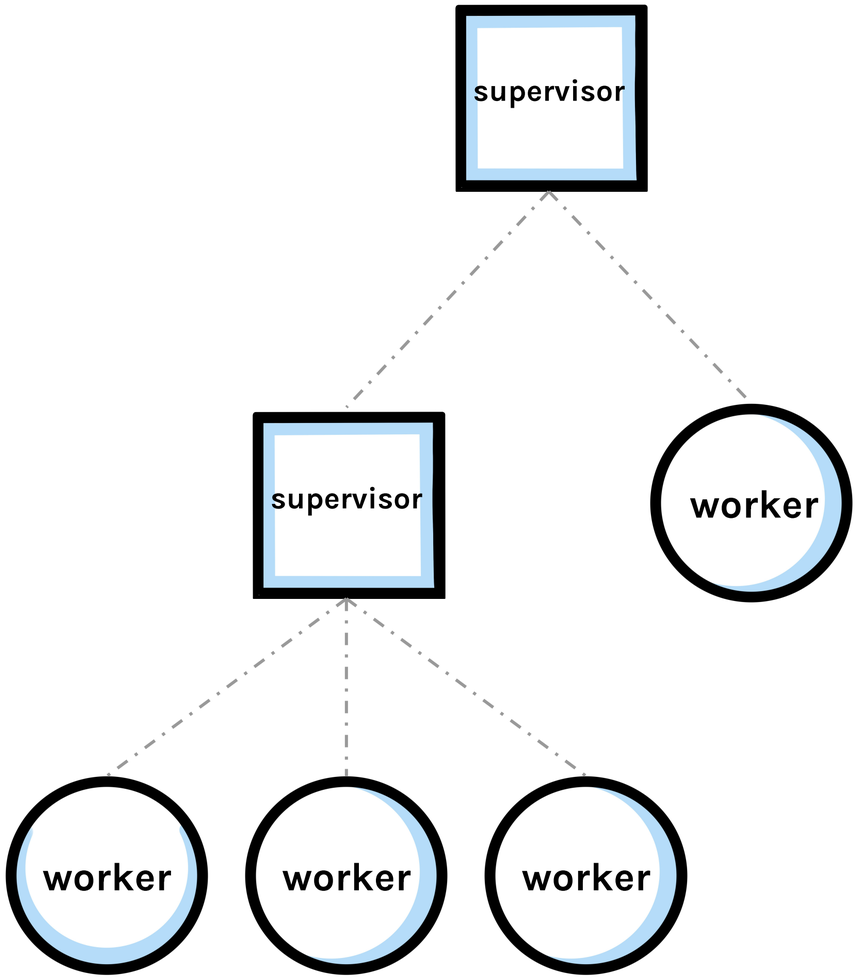

```
Ref = erlang:monitor(process, PidB)
```

- **erlang:monitor/2** will the process calling the BIF monitor the process **PidB.** The monitoring is uni-directional and stackable.
- When process PidB dies then PidA will be sent the message **{'DOWN', Ref, process, PidB, Reason}**
- A monitor can be turned by calling **erlang:demonitor(Ref)** and **erlang:demonitor(Ref, [flush])**

Robust Systems

- Building a system in layers can make it robust
  - Level N-1 traps and fixes errors occurring in level N
  - The leaves of the tree are workers
- In well designed systems, application programmers will not have to worry about error handling code
  - Error handling will be isolated by higher levels of the system, managed uniformly across processes
- Processes whose only task is to supervise children are called supervisors

Robust Systems



- Robust systems can be designed by layering

A Robust Server

Remember the server example in the process design patterns section? *The Server is unreliable!*

- Let's rewrite the server making it reliable by monitoring the clients
- Let’s rewrite the clients making them reliable by monitoring the server

A Robust Server

```
-module(frequency).
-export([start/0, stop/0,  allocate/0, deallocate/1]).
-export([init/0]).

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    process_flag(trap_exit, true),
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

get_frequencies() -> [10,11,12,13,14, 15].
```

A Server Example

```
stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message protocol in
%% functional interfaces.

call(Message) ->
    Ref = erlang:monitor(process, frequency),
    frequency ! {request, self(), Message},
    receive
       {reply, Reply} ->
          erlang:demonitor(Ref, [flush]),
          Reply;
       {'DOWN', Ref, process, Pid, Reason} ->
          exit(Reason)
    end.

reply(Pid, Message) -> Pid ! {reply, Message}.
```

Keep race conditions in mind!

A Server Example

```
%% The main server loop.

loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
       {NewFrequencies,Reply} = allocate(Frequencies,Pid),
       reply(Pid, Reply),
       loop(NewFrequencies);
   {request, Pid , {deallocate, Freq}} ->
       NewFrequencies = deallocate(Frequencies, Freq),
       reply(Pid, ok),
       loop(NewFrequencies);
   {request, Pid, stop} ->
       reply(Pid, ok);
   {'EXIT', Pid, Reason} ->
       NewFrequencies = exited(Frequencies, Pid),
       loop(NewFrequencies)
 end.
```

A Robust Server

```
%% The Internal Functions
%% Functions used to allocate and deallocate frequencies.

allocate({[], Allocated}, Pid) ->
    {{[], Allocated}, {error, no_frequencies}};
allocate({[Freq|Frequencies], Allocated}, Pid) ->
    link(Pid),
    {{Frequencies, [{Freq, Pid}|Allocated]}, {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    {Freq, Pid} = lists:keyfind(Freq, 1, Allocated),
    unlink(Pid),
    NewAllocated = lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free],  NewAllocated}.
```

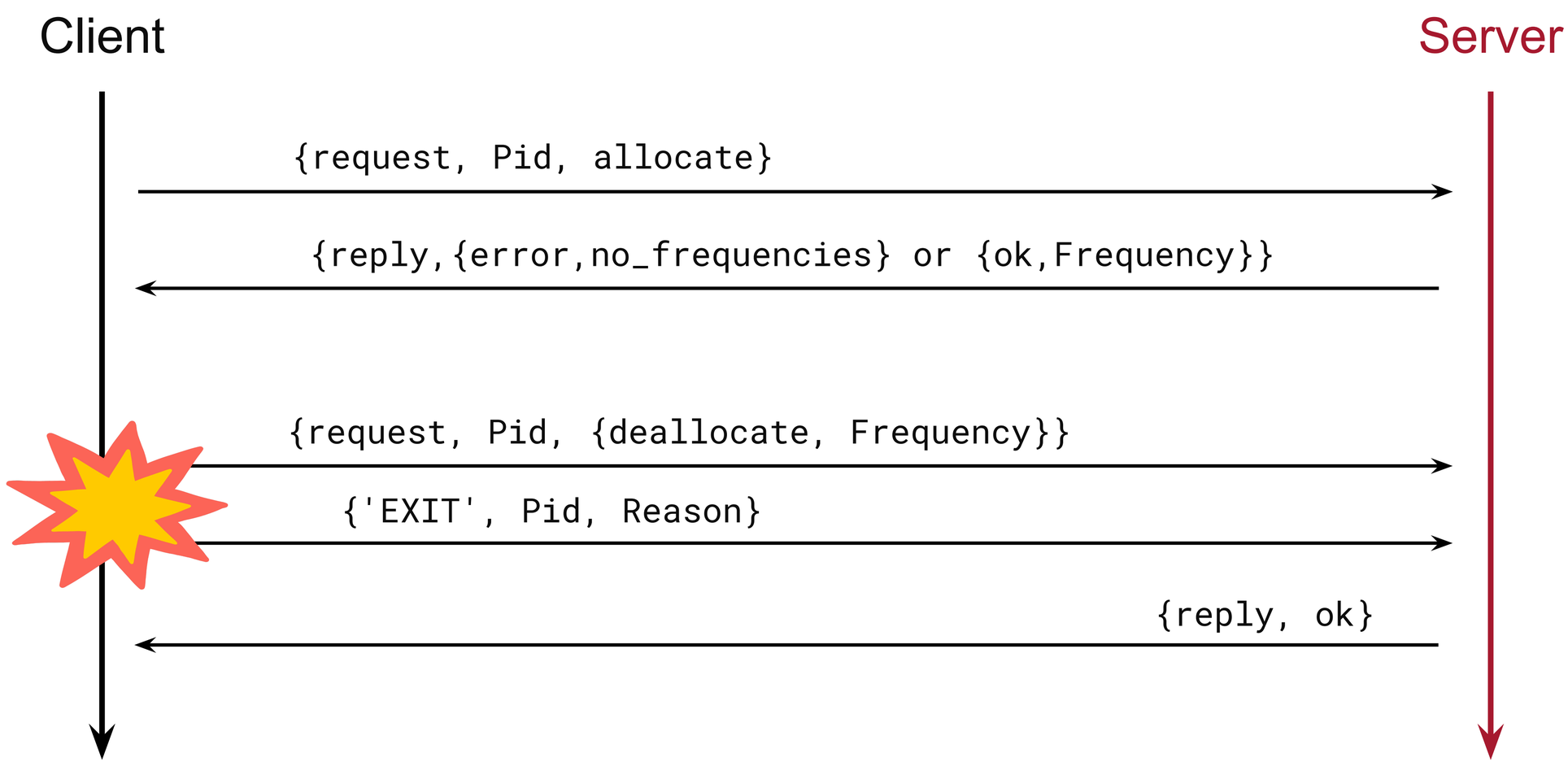
A Robust Server

```
%% Help function used when a client crashes.

exited({Free, Allocated}, Pid) ->
  case lists:keyfind(Pid, 2, Allocated) of
    {Freq, Pid} ->
      NewAllocated = lists:keydelete(Freq, 1, Allocated),
      {[Freq|Free],  NewAllocated};
    false ->
      {Free, Allocated}
  end.
```

Keep race conditions in mind!

A Server Example



Process Error Handling

- Links
- Exit Signals
- Definitions
- Propagation Semantics
- Monitors
- Robust Systems
- Error Handling Example