Advanced Programming

Erlang Introduction — The Sequel

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

October 5, 2017

Today's Menu

- Recap
- Designing Servers
- Background: Registering processes
- Background: Exceptions again
- Linking processes
- Library code for making robust servers

Recap

- Organise your code in modules
- Make sure that you understand the basic concurrency primitives in Erlang
- Review the `fib` and `cfib` functions in the `fib` module.

Dealing With State

- Functions are pure (stateless).
- Processes are stateful.
- We organise our code as micro-servers that manage a state which can be manipulated via a client API.
- Functions starts processes, processes runs functions, functions are defined in modules.

Client–Server Basic Request-Response

- A server is process that loops (potentially) forever.
- Clients communicate through a given API/Protocol
- That is, we start with the following template:

```
start() -> spawn(fun () -> loop(Initial) end).
request_reply(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} -> Response
    end.
loop(State) ->
    receive
        {From, Request} ->
            {NewState, Res} = ComputeResult(Request, State),
            From ! {self(), Res},
            loop(NewState)
    end.
```

Example: Position Server

```
start(Start) -> spawn(fun () -> loop(Start) end).
move(Pid, Dir) -> request_reply(Pid, {move, Dir}).
get_pos(Pid) -> request_reply(Pid, get_pos).
request_reply(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} -> Response
    end.
loop({X,Y}) ->
    receive
        {From, {move, north}} ->
            From ! {self(), ok},
            loop({X, Y+1});
        {From, {move, west}} ->
            From !  {self(), ok},
            loop({X-1, Y});
        {From, {get_pos}} ->
            From ! {self(), {X,Y}},
            loop({X,Y})
    end.
```

Student activity: Count Server

- Let’s make a server that can keep track of a counter
- What is the client API?
- What is the internal state?

Example: Phone-Book, Interface

```
start() -> spawn(fun() -> loop(#{}) end).

add(Pid, Contact) ->
    request_reply(Pid, {add, Contact}).

list_all(Pid) ->
    request_reply(Pid, list_all).

update(Pid, Contact) ->
    request_reply(Pid, {update, Contact}).
```

Example: Phone-Book, Impletemtation 1

```
loop(Contacts) ->
    receive
        {From, {add, Contact}} ->
            {Name,_,_} = Contact,
            case maps:is_key(Name, Contacts) of
                false ->
                    From ! {self(), ok},
                    loop(Contacts#{Name => Contact});
                true ->
                    From ! {self(), {error, Name, is_already_there}},
                    loop(Contacts)
            end;
```

Example: Phone-Book, Implementation 2

```
        {From, list_all} ->
            List = maps:to_list(Contacts),
            From ! {self(),
                    {ok, lists:map(fun({_, C}) -> C end, List)}},
            loop(Contacts);
        {From, {update, Contact}} ->
            {Name,_,_} = Contact,
            NewContacts = maps:remove(Name, Contacts),
            From ! {self(), ok},
            loop(maps:put(Name, Contact, NewContacts));
        {From, Other} ->
            From ! {self(), {error,unknow_request, Other}},
            loop(Contacts)
    end.
```

Communication Patterns

- Synchronous (aka Blocking), like the simple Request-Response function `blocking` (aka `request_reply`).

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

- Asynchronous (aka Non-Blocking), standard sending of messages in Erlang

```
async(Pid, Msg) ->
   Pid ! Msg.
```

Design “Method”

- Determine the API:
  - names
  - types
  - blocking or non-blocking
- Design internal protocols
- Split into servers (processes)

Design Task

You are an intern at MicroCorp. Your team has decided that you need a new logging mechanism which potentially can be distributed across machines (thus Erlang is the tool of choice). To start with, the logging mechanism should have basic functionality: it should be possible to log something (`log`), and it should be possible to retrieve what has been logged (`retrieve`).

What is your design?

Background: Registering Processes

- It can be convenient to register a process under a global name, so that we can easily get it without threading a pid around.
- `register(Name, Pid)` registers the process with `Pid` under `Name`
- `whereis(Name)` gives us the pid registered for `Name`; or `undefined` if `Name` is not registered
- `Name ! Mesg` sends `Mesg` to the process registered under `Name`.

Background: Exceptions Revisited

Exceptions comes in different flavours:

```
try Expr of
  Pat1 -> Expr1;
  Pat2 -> Expr2;
  ...
catch
  ExType1: ExPat1 -> ExExpr1;
  ExType2: ExPat2 -> ExExpr2;
  ...
after
  AfterExpr
end
```

Where `ExType` is either `throw`, `exit`, or `error` (`throw` is the default).

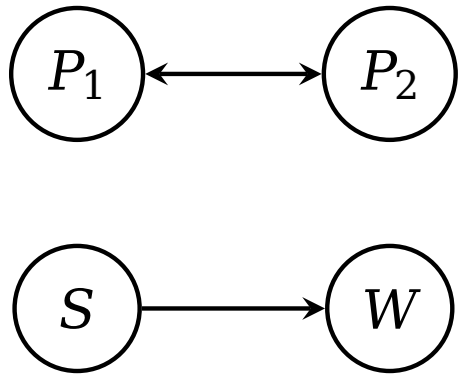
Background: Generating Exceptions

- We can generate all kind of exceptions:
  - `throw(Why)` to throw an exception that the caller might want to catch. For normal exceptions.
  - `exit(Why)` to exit the current process. If not caught then the message `{'EXIT',Pid,Why}` is broadcast to all *linked* processes.
  - `erlang:error(Why)` for internal errors, that nobody is really expected to handle.
- Thus, to catch *all* exceptions we need the following pattern:

```
try Expr
catch
  _ : _ -> MightyHandler
end
```

Robust Systems

- We need at least two computers(/nodes/processes) to make a robust system: one computer(/node/process) to do what we want, and one to monitor the other and take over when errors happens.
- `link(Pid)` makes a symmetric link between the calling process and `Pid`.
- `monitor(process, Pid)` makes an asymmetric link between the calling process and `Pid`.



Linking Processes

- If we want to handle when a linked process crashes then we need to call `process_flag(trap_exit, true)`.
- Thus, we have the following idioms for creating processes:
  - Idiom 1, I don't care:

    ```
    Pid = spawn(fun() -> ... end)
    ```

  - Idiom 2, I won't live without her:

    ```
    Pid = spawn_link(fun() -> ... end)
    ```

  - Idiom 3, I'll handle the mess-ups:

    ```
    process_flag(trap_exit, true),
    Pid = spawn_link(fun() -> ... end),
    loop(...).

    loop(State) ->
      receive
        {'EXIT', Pid, Reason} -> HandleMess, loop(State);
        ...
      end
    ```

Example: Keep ~~Trucking~~ Looping

Suppose that we really most have a phonebook server running at all times. How do we monitor the phonebook server and restart it if (when?) it crashes.

Example: Keep Looping

```
start() -> keep_looping().
request_reply(Pid, Request) -> Ref = make_ref(),
                      Pid ! {self(), Ref, Request},
                      receive {Ref, Response} -> Response end.
keep_looping() ->
    spawn(fun () ->
            process_flag(trap_exit, true),
            Worker = spawn_link(fun() -> loop(#{}) end),
            supervisor(Worker)
          end).
supervisor(Worker) ->
    receive
        {'EXIT', Pid, Reason} ->
            io:format("~p exited because of ~p~n", [Pid,Reason]),
            Pid1 = spawn_link(fun() -> loop(#{}) end),
            supervisor(Pid1);
        Msg -> Pid ! Msg, supervisor(Pid)
    end.
```

Distributed Programs in Erlang

- *Distributed Erlang* for tightly coupled computers in a secure environment.
  - `spawn(Node, Fun)` to spawn a process running `Fun` on `Node`
  - `{RegAtom, Node} ! Mesg` sends `Mesg` to the process registered as `RegAtom` at `Node`.
  - `monitor_node(Node, Flag)` register the calling process for notification about `Node` if `Flag` is `true`; if `Flag` is `false` then monitoring is turned off.
- *Sockets* for untrusted environments:
  - To build a middleware layer for Erlang nodes
  - For inter-language communication.

  See the documentation for `gen_tcp` and `gen_udp`

Setting Up Some Erlang Nodes

- To start nodes on the same machine, start `erl` with option `-sname`
- To start nodes on different machines, start `erl` with options `-name` and `-setcookie`:
  - On machine A:

    ```
    erl -name bart -setcookie BoomBoomShakeTheRoom
    ```

  - On machine B:

    ```
    erl -name homer -setcookie BoomBoomShakeTheRoom
    ```

- `rpc:call(Node, Mod, Fun, Args)` evaluates `Mod:Fun(Args)` on `Node`. (See the the manual page for `rpc` for more information.)

Generic Servers

- Goal: Abstract out the difficult handling of concurrency to a generic library
- The difficult parts:
  - The `start-request_reply/async-loop` pattern
  - Supervisors

Basic Server Library

```
start(Name, Mod) ->
  register(Name, spawn(fun() -> loop(Name, Mod, Mod:init())
                                end)).
request_reply(Pid, Request) ->
  Pid ! {self(), Request},
  receive
    {Pid, Reply} -> Reply
  end.
loop(Name, Mod, State) ->
  receive
      {From, Request} ->
          {Reply, State1} = Mod:handle(Request, State),
          From ! {Name, Reply},
          loop(Name, Mod, State1)
  end.
```

Example: Phonebook Callback Module, 1

```
-module(pb).
-import(basicserver, [request_reply/2]).

%% Interface
start()          -> basicserver(phonebook, pb).
add(Contact)     -> request_reply(pb, {add, Contact}).
list_all()       -> request_reply(pb, list_all).
update(Contact) -> request_reply(pb, {update, Contact}).
```

Example: Phonebook Callback Module, 2

```
%% Callback functions
init() -> #{}.

handle({add, {Name, _, _} = Contact}, Contacts) ->
    case maps:is_key(Name, Contacts) of
        false -> {ok, Contacts#{Name => Contact}};
        true  -> {{error, Name, is_already_there},
                  Contacts}
    end;
handle(list_all, Contacts) ->
    List = maps:to_list(Contacts),
    {{ok, lists:map(fun({_, C}) -> C end, List)},
     Contacts};
handle({update, {Name, _, _} = Contact}, Contacts) ->
    {ok, Contacts#{Name => Contact}}.
```

Say something about Kaboose!

Summary

- How design micro-server: blocking vs non-blocking
- To make a robust system we need two parts: one to do the job and one to take over in case of errors
- Structure your code into the infrastructure parts and the functional parts.