Advanced Programming

Erlang Introduction — The Sequel

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

September, 2021

Part I

Pre-lecture – Stateful Servers

Dealing With State

- Unprotected shared mutable state is bad
- OOP: State should protected in objects, and only manipulated via a given interface (API)
- FP: Ban mutation, problem solved
  - If (when) we need state, we'll just pass (part of) the world around
  - which is clunky …MONADS to the rescue

Dealing With State in Erlang

- 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 (a.k.a concurrent objects).
- 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: Count Server – Client API

```
-module(counter).

-export([start/0, incr/1, decr_with/2, get_value/1]).

start()            -> spawn(fun () -> loop(0) end).
incr(Cid)          -> request_reply(Cid, increment).
decr_with(Cid, N)  -> request_reply(Cid, {decr, N}).
get_value(Cid)     -> request_reply(Cid, get_value).

request_reply(Pid, Request) -> ...
```

Example: Count Server – Server loop

```
loop(Count) ->
    receive
      {From, increment} ->
        {NewState, Res} = {Count + 1, ok},
        From ! {self(), Res},
        loop(NewState);
      {From, {decr, N}} ->
        {NewState, Res} = {Count - N, ok},
        From ! {self(), Res},
        loop(NewState);
      {From, get_value} ->
        {NewState, Res} = {Count, {ok, Count}},
        From ! {self(), Res},
        loop(NewState)
    end.
```

Example: Count Server – Using the API

```
1> c("counter.erl").             % Compile counter.erl
c("counter.erl").
{ok,counter}
2> C1 = counter:start().         % Start a count server
C1 = counter:start().
<0.86.0>
3> counter:incr(C1).             % Increment the server's value
counter:incr(C1).
ok
4> counter:decr_with(C1, 43).   % Decrement the server's value
counter:decr_with(C1, 43).
ok
5> counter:get_value(C1).        % Get the current value
counter:get_value(C1).
{ok,-42}
```

Today's Menu

- Recap
  - Stateful servers
  - Background: Registering processes
  - Background: Exceptions again
- Multi-process servers
- Robust systems
- Separation of concerns

Part II

Recap – Stateful Servers

Recap

- Organise your code in modules
- Make sure that you understand the basic concurrency primitives in Erlang
- 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 (a.k.a concurrent objects).
- 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

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: Monster Server – Client API

```
start(Pos, Hp)    -> spawn(fun () -> loop({Pos, Hp}) end).
move(Mid, Dir)    -> request_reply(Mid, {move, Dir}).
get_pos(Mid)      -> request_reply(Mid, get_pos).
damage(Mid, Dmg) -> request_reply(Mid, {hit, Dmg}).

request_reply(Pid, Request) -> ...
```

Example: Monster Server – Server loop

```
loop({{X, Y}, Health} = State) ->
    receive
        {From, {move, north}} ->
            {NewState, Res} = {{{X, Y+1}, Health}, ok},
            From ! {self(), Res},
            loop(NewState);
        {From, {move, west}} ->
            {NewState, Res} = {{{X-1, Y}, Health}, ok},
            From ! {self(), Res},
            loop(NewState);
        {From, get_pos} ->
            {NewState, Res} = {State, {ok, {X,Y}}},
            From ! {self(), Res},
            loop(NewState);
        {From, {hit, Dmg}} ->
            NewHealth = Health-Dmg,
            if NewHealth > 0 ->
                    From ! {self(), ouch},
                    loop({{X,Y}, NewHealth});
               NewHealth =< 0 ->
                    From ! {self(), dead}
            end
    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

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

Design “Method”

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

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: 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`.

Warning: On the exam you should normally **not** register processes without being asked to do so explicitly (just like with timeouts).

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
```

Part III

Multi-Process Servers

Multi-Process Servers

- Sometimes it make sense to use multiple processes to implement a server. It's often transparent for the clients.
- Two major use-cases:
  - We have a long running function, and we don't want to block other clients (*latency*).
  - We have an API function that takes a client-supplied function as argument, that should be executed by the server.

Long running functions – Improve latency

- Assume we have an API function, `slow`, that might have a long running time.
- If the server only consists of one process, and the process compute the result of `slow`, then we cannot serve other clients meanwhile.

Long running functions – Improve latency

- Assume we have an API function, `slow`, that might have a long running time.
- If the server only consists of one process, and the process compute the result of `slow`, then we cannot serve other clients meanwhile.
- Solution: Start a new (worker) process to compute the result. While the worker is working, the (main) server process can serve other clients.

Long running functions – Improve latency

- Assume we have an API function, `slow`, that might have a long running time.
- If the server only consists of one process, and the process compute the result of `slow`, then we cannot serve other clients meanwhile.
- Solution: Start a new (worker) process to compute the result. While the worker is working, the (main) server process can serve other clients.
- Complications:
  - Who should send the response of `slow` to the client?
  - Consistency

Protocol Variation

- If we don’t know who is going to send a reply to the client, we might want to use an alternative request-reply protocol.

Protocol Variation

- If we don't know who is going to send a reply to the client, we might want to use an alternative request-reply protocol.
- Client:

```
request_reply(Pid, Request) ->
  Ref = make_ref(),
  Pid ! {self(), Ref, Request},
  receive
    {Ref, Response} -> Response
  end.
```

Protocol Variation

- If we don't know who is going to send a reply to the client, we might want to use an alternative request-reply protocol.
- Client:

```
request_reply(Pid, Request) ->
  Ref = make_ref(),
  Pid ! {self(), Ref, Request},
  receive
    {Ref, Response} -> Response
  end.
```

- Server:

```
loop(State) ->
  receive
    {From, Ref, Request} ->
      {NewState, Res} = ComputeResult(Request, State),
      From ! {Ref, Res},
      loop(NewState)
    ...
  end.
```

Functions as API Arguments

- Assume we have an API function that takes a function as an argument: `danger(S, Fun)`
- If the server only consists of one process, and that process should execute `Fun`, then we don't know what can happen.
  We are crossing a *trust barrier*.

Functions as API Arguments

- Assume we have an API function that takes a function as an argument: `danger(S, Fun)`
- If the server only consists of one process, and that process should execute `Fun`, then we don't know what can happen.
  We are crossing a *trust barrier*.
- Solution: Start a new (worker) process to compute the result of running `Fun`. (*isolation*)

Functions as API Arguments

- Assume we have an API function that takes a function as an argument: `danger(S, Fun)`
- If the server only consists of one process, and that process should execute `Fun`, then we don't know what can happen.
  We are crossing a *trust barrier*.
- Solution: Start a new (worker) process to compute the result of running `Fun`. (*isolation*)
- Complications:
  - We often need to make some assumptions about `Fun`
  - Various failure classes from `Fun`: throws, exits, errors or fail to terminate.
  - Consistency

Timeouts

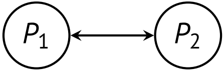
- A crude way to deal with functions that does not terminate is to set a time limit for the execution (a.k.a a *timeout*).
- However, if the specification/domain does *not* give you a time limit then **you can only pick the wrong timeout**.
- (Strictly enforced in this course)
- (Can still be useful during development)

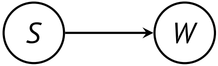
Part IV

Robust Systems

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`. Often we use the `spawn_link` function that spawns a new process and link it.
- `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)
```

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 them:

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

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 them:

    ```
    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: Advanced Counting Server – New API

- Extend the API with two functions:

  ```
  get_slow(Cid)       -> request_reply(Cid, get_slow).
  modify(Cid, Fun)    -> request_reply(Cid, {modify, Fun}).
  ```

  Where:

  - `get_slow(Cid)` gets the value of `Cid` via a slow computation.
  - `modify(Cid, Fun)` modifies the value of `Cid` by applying the function `Fun` to the internal state.

Example: Advanced Counting Server – First implementation

```
loop(Count) ->
  receive
    ...
    {From, get_slow} ->
      {NewState, Res} = slow_value(Count),
      From ! {self(), Res},
      loop(NewState);
    {From, {modify, Fun}} ->
      {NewState, Res} = compute_modified(Count, Fun),
      From ! {self(), Res},
      loop(NewState)
  end.
slow_value(Count) -> ...
compute_modified(Count, Fun) ->
  New = Fun(Count),
  {New, ok}.
```

Example: Advanced Counting Server – Slow Function

```
request_reply(Pid, Request) ->
    Ref = make_ref(),
    Pid ! {self(), Ref, Request},
    receive
        {Ref, Response} -> Response
    end.
loop(Count) ->
  receive
    ...
    {From, Ref, get_slow} ->
      _Worker = spawn(fun () ->
                           {_, Res} = slow_value(Count),
                           From ! {Ref, Res}
                       end),
      NewState = Count,
      loop(NewState)
  end.
```

Example: Advanced Counting Server – Isolation

```
loop(Count) ->
  receive ...
    {From, Ref, {modify, Fun}} ->
      Me = self(),
      process_flag(trap_exit, true),
      Worker =
        spawn_link(fun() ->
                     {NewState, Res} = compute_modified(Count, Fun),
                     From ! {Ref, Res},
                     Me ! {self(), NewState}
                  end),
      NewState = receive
                   {Worker, New} -> New;
                   {'EXIT', Worker, Reason} ->
                     From ! {Ref, {error, Reason}},
                     Count
                 end,
      loop(NewState)
  end.
```

Separation of Concerns

```
handle_call(Request, Count) ->
  case Request of
    increment     -> {Count + 1, ok};
    {decr, N}     -> {Count - N, ok};
    get_value     -> {Count, {ok, Count}};
    get_slow      -> slow_value(Count);
    {modify, Fun} -> compute_modified(Count, Fun)
  end.

loop(State) ->
  receive
    {From, Ref, Request} ->
      {NewState, Res} = handle_call(Request, State),
      From ! {Ref, Res},
      loop(NewState)
  end.
```

(Adding support for worker processes is left as an exercise.)

Part V

Summary

Emoji Assignment

Remember to test

```
-module(test_emoji).
-include_lib("eunit/include/eunit.hrl").
-export([test_all/0]).
test_all() -> eunit:test(testsuite(), [verbose]).
testsuite() ->
    [ {"Basic behaviour", spawn,
       [ test_start_server()
       , test_shortcode_smiley() ] } ].
test_start_server() ->
    {"We can call start/1 and it does not crash",
     fun () ->
       ?assertMatch({ok, _}, emoji:start([]))
     end }.
test_shortcode_smiley() ->
    {"Register new shortcode",
     fun () ->
       {ok, S} = emoji:start([]),
       ?assertEqual(ok, emoji:new_shortcode(S, "smiley",
                                            <<240,159,152,131>>))
     end }.
```

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.
- This week's assignment: Emoji galore