Advanced Programming

Riding the OTP

Mikkel Kragh Mathiesen
mkm@di.ku.dk

Department of Computer Science
University of Copenhagen

October, 2020

Slides by Ken Friis Larsen

Today's Menu

- Library code for making robust servers, the last piece
- Open Telecom Platform (OTP)
- How to program IoT devices: intelligent lamps and doors

Part I

Generic Servers – Behaviours

Generic Servers

- Goal: Abstract out the difficult handling of concurrency to a generic library
- The difficult parts:
  - The `start-request_reply(/nonblocking)-loop` pattern
  - Supervisors
  - Hot-swapping of code

Hot Code Swapping

```
swap_code(Name, Mod) -> request_reply(Name, {swap_code, Mod}).
request_reply(Pid, Request) ->
    Pid ! {self(), Request},
    receive {Pid, Reply} -> Reply
    end.
loop(Name, Mod, State) ->
    receive
        {From, {swap_code, NewMod}} ->
            From ! {Name, ok},
            loop(Name, NewMod, State);
        {From, Request} ->
            {Reply, State1} = Mod:handle(Request, State),
            From ! {Name, Reply},
            loop(Name, Mod, State1)
    end.
```

Example: Counter Callback Module, 1

```
-module(server_counter).

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

% Server behaviour
-export([init/0, handle_request/2, handle_request_reply/2]).
-behaviour(server).

%% Interface
start() -> server:start(?MODULE).
incr(Cid) -> server:request(Cid, incr), ok.
decr_with(Cid, N) -> server:request_reply(Cid, {decr_with, N}).
get_value(Cid) -> server:request_reply(Cid, get_value).
```

Example: Counter Callback Module, 2

```
%% Callback functions
init() -> 0.

handle_request(incr, Count) ->
    Count + 1.

handle_request_reply({decr_with, N}, Count) ->
    {Count - N, ok};
handle_request_reply(get_value, Count) ->
    {Count, {ok, Count}}.
```

What is the behaviour

- For a callback module to work with `server` it needs to export three functions `init`, `handle_request` and `handle_request_reply`.
  It would be great if someone could help us get that right…

What is the behaviour

- For a callback module to work with `server` it needs to export three functions `init`, `handle_request` and `handle_request_reply`.
  It would be great if someone could help us get that right…
- The compiler can help us

What is the behaviour

- For a callback module to work with `server` it needs to export three functions `init`, `handle_request` and `handle_request_reply`.
  It would be great if someone could help us get that right…
- The compiler can help us
- In `server.erl`:

```
-callback init() -> State :: term().
-callback handle_request(Msg :: term(), State :: term())
    -> State :: term().
-callback handle_request_reply(Msg :: term(), State :: term()
    -> {State :: term(), Response :: term()}.
                    { Reply :: any(), State :: S }.
```

What is the behaviour

- For a callback module to work with `server` it needs to export three functions `init`, `handle_request` and `handle_request_reply`.
  It would be great if someone could help us get that right…
- The compiler can help us
- In `server.erl`:

```
-callback init() -> State :: term().
-callback handle_request(Msg :: term(), State :: term())
    -> State :: term().
-callback handle_request_reply(Msg :: term(), State :: term()
    -> {State :: term(), Response :: term()}.
                    { Reply :: any(), State :: S }.
```

- In `server_counter.erl`:

```
-behaviour(server).
```

Part II

OPT – Supervisors & State Machines

Open Telecom Platform (OTP)

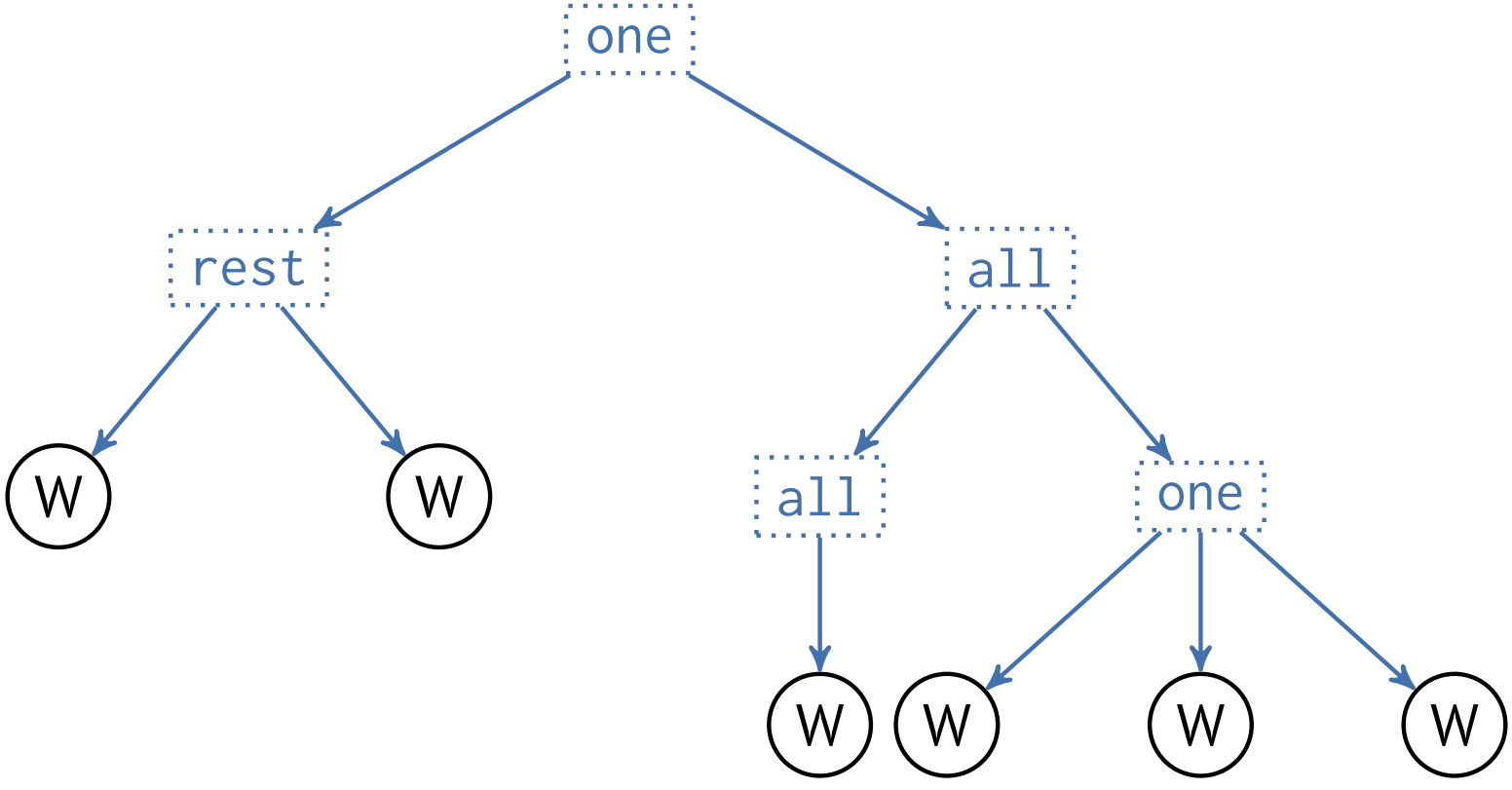
- Library(/framework/platform) for building large-scale, fault-tolerant, distributed applications.
- A central concept is the OTP *behaviour*
- Some behaviours
  - `supervisor`
  - `gen_server`
  - `gen_statem` (or `gen_fsm`)
  - `gen_event`
- See `proc_lib` and `sys` modules for basic building blocks.

Using gen_server

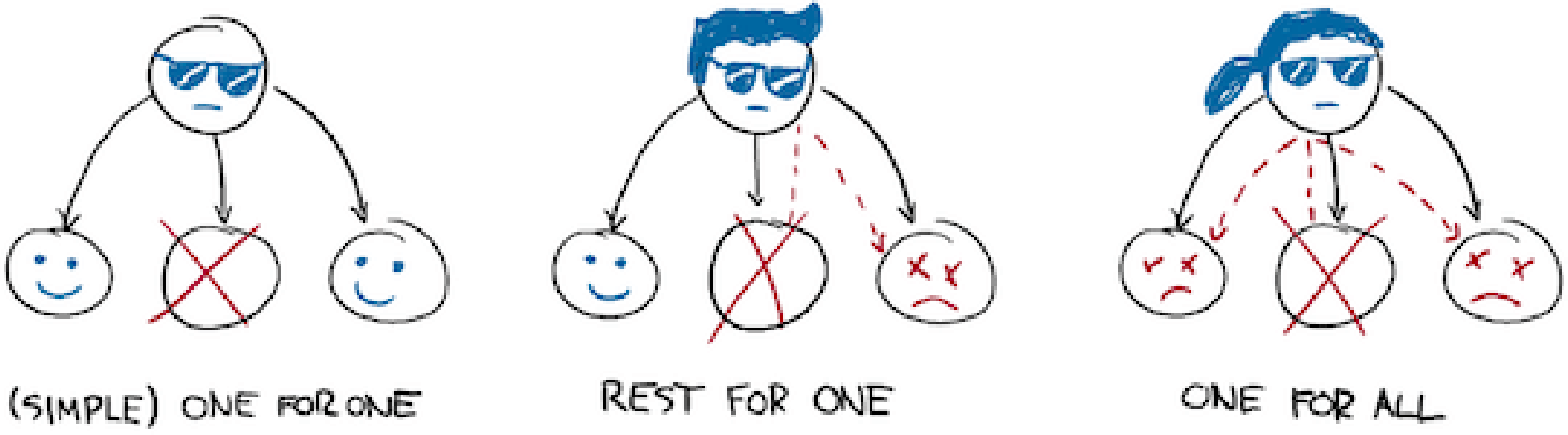
- Step 1: Decide module name
- Step 2: Write client interface functions
- Step 3: Write the six server callback functions:
  - `init/1`
  - `handle_call/3`
  - `handle_cast/2`
  - `handle_info/2`
  - `terminate/2`
  - `code_change/3`

  (you can implement the callback functions by need.)

Supervisor Trees



Supervisors Restart Strategies

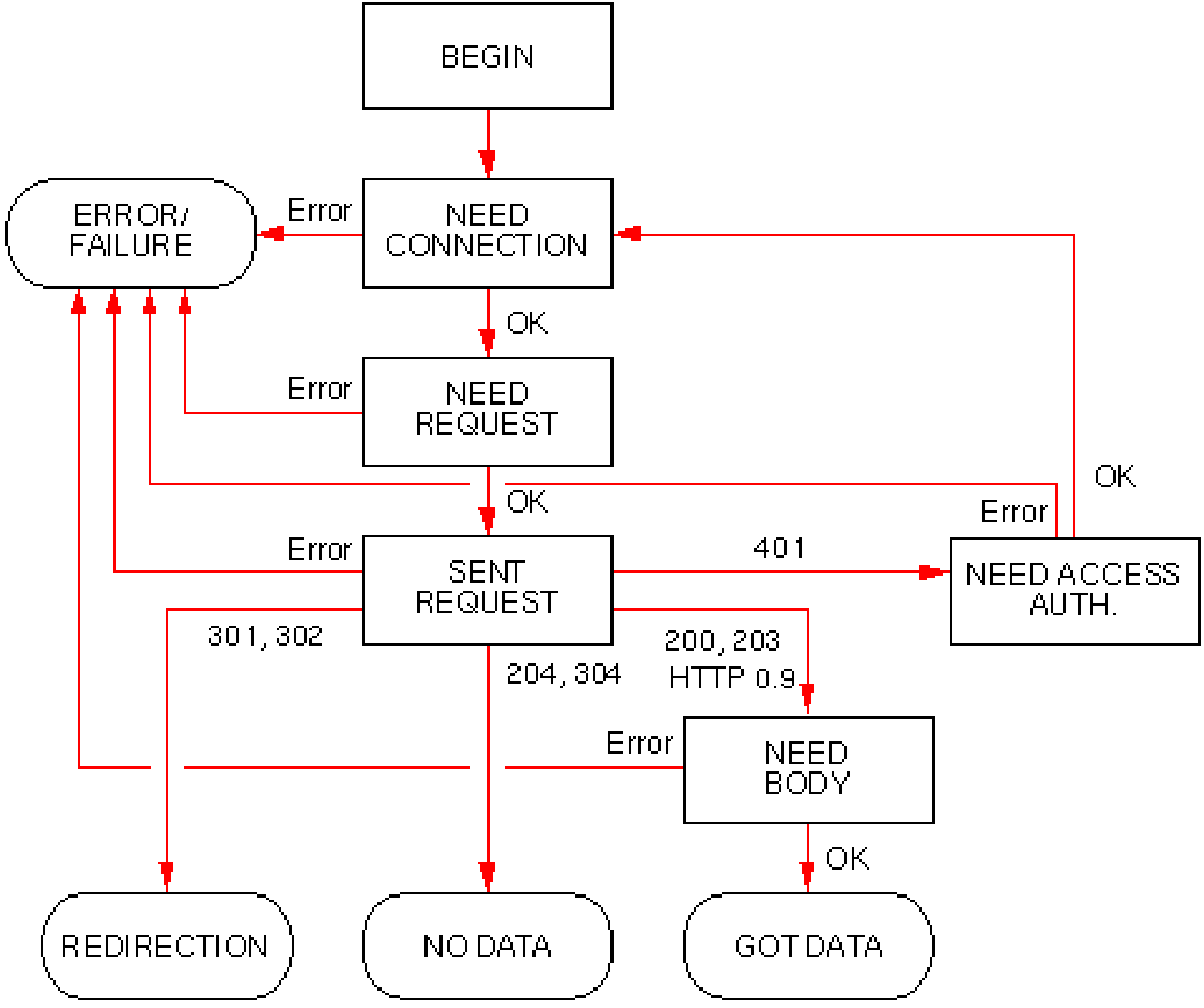


(Image credit Ferd Hebert)

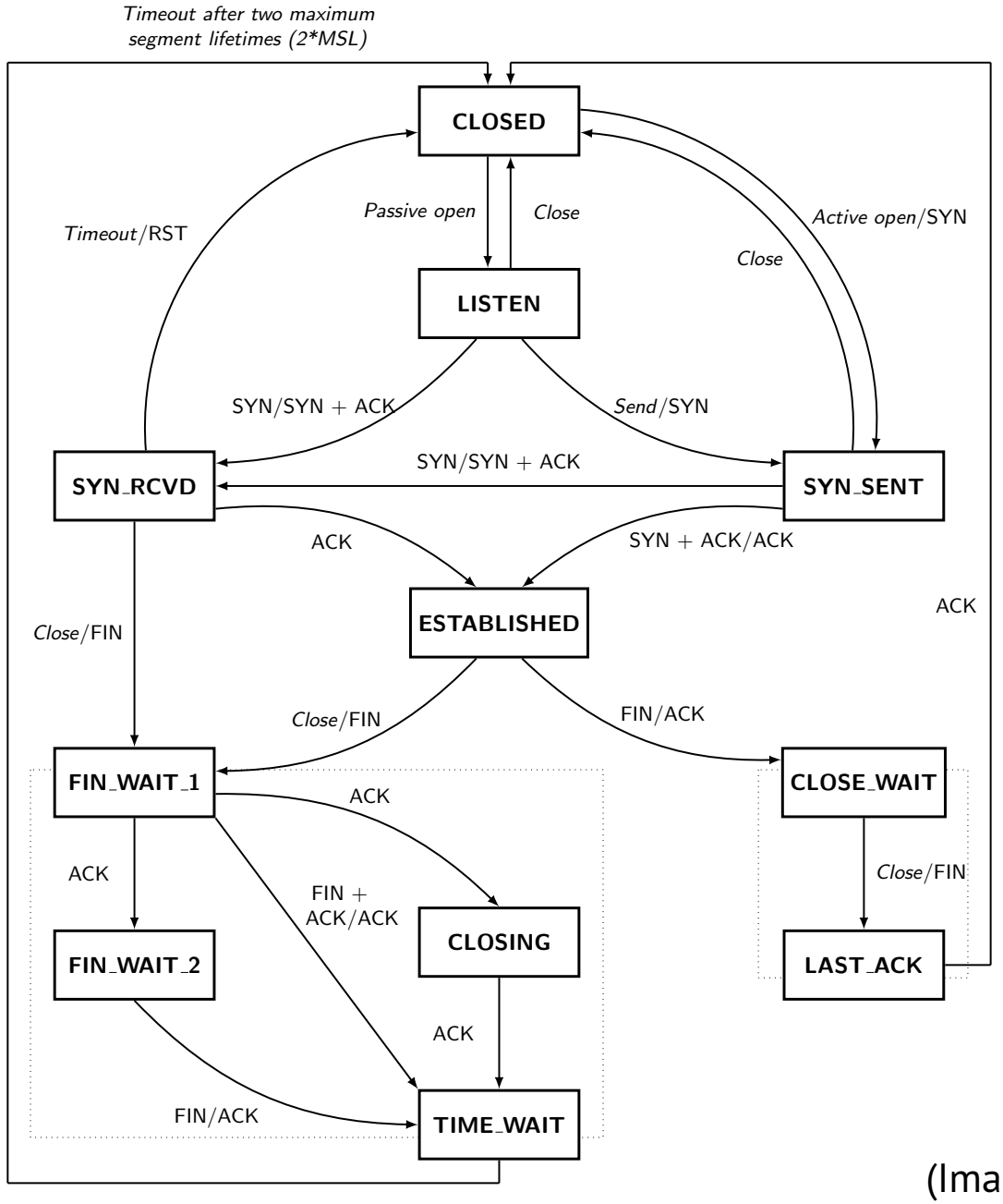
Using `gen_statem`

- Step 1: Decide module name
- Step 2: Write client interface functions
- Step 3: Write following callback functions:
  - `init/1`
  - `callback_mode/0` should return `state_functions` or `handle_event_function`
  - `terminate/3`
  - `code_change/4`
  - `handle_event/4` or some `StateName/3` functions

HTTP Client State Machine



TCP (RFC 793) State Machine



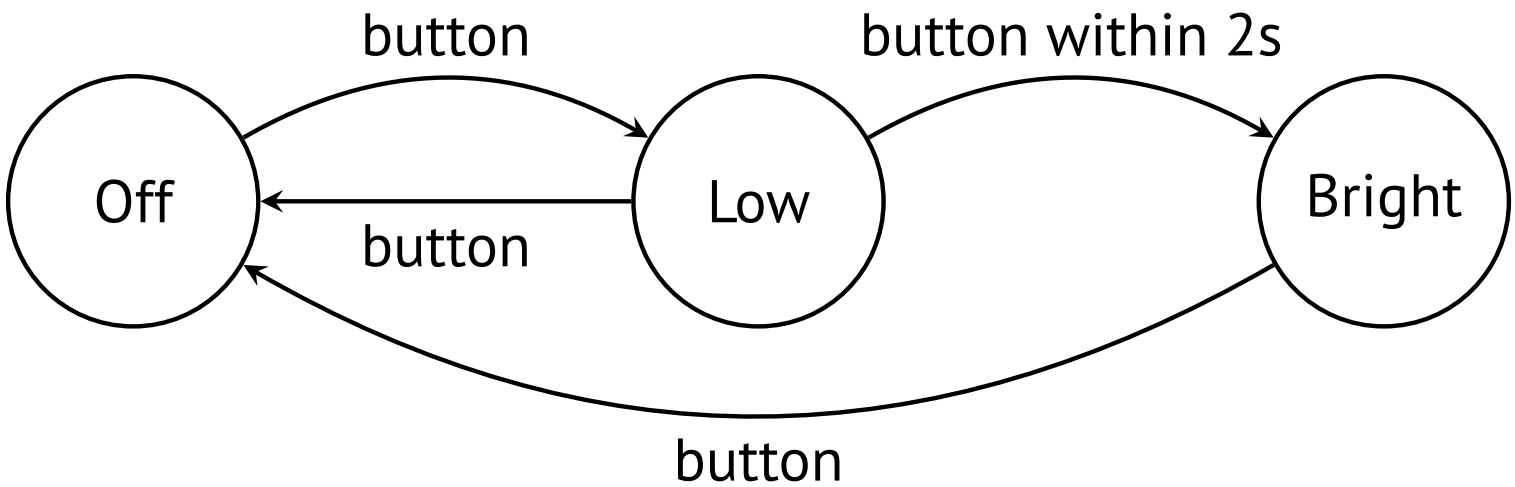
(Image credit Ivan Griffin)

A fancy lamp

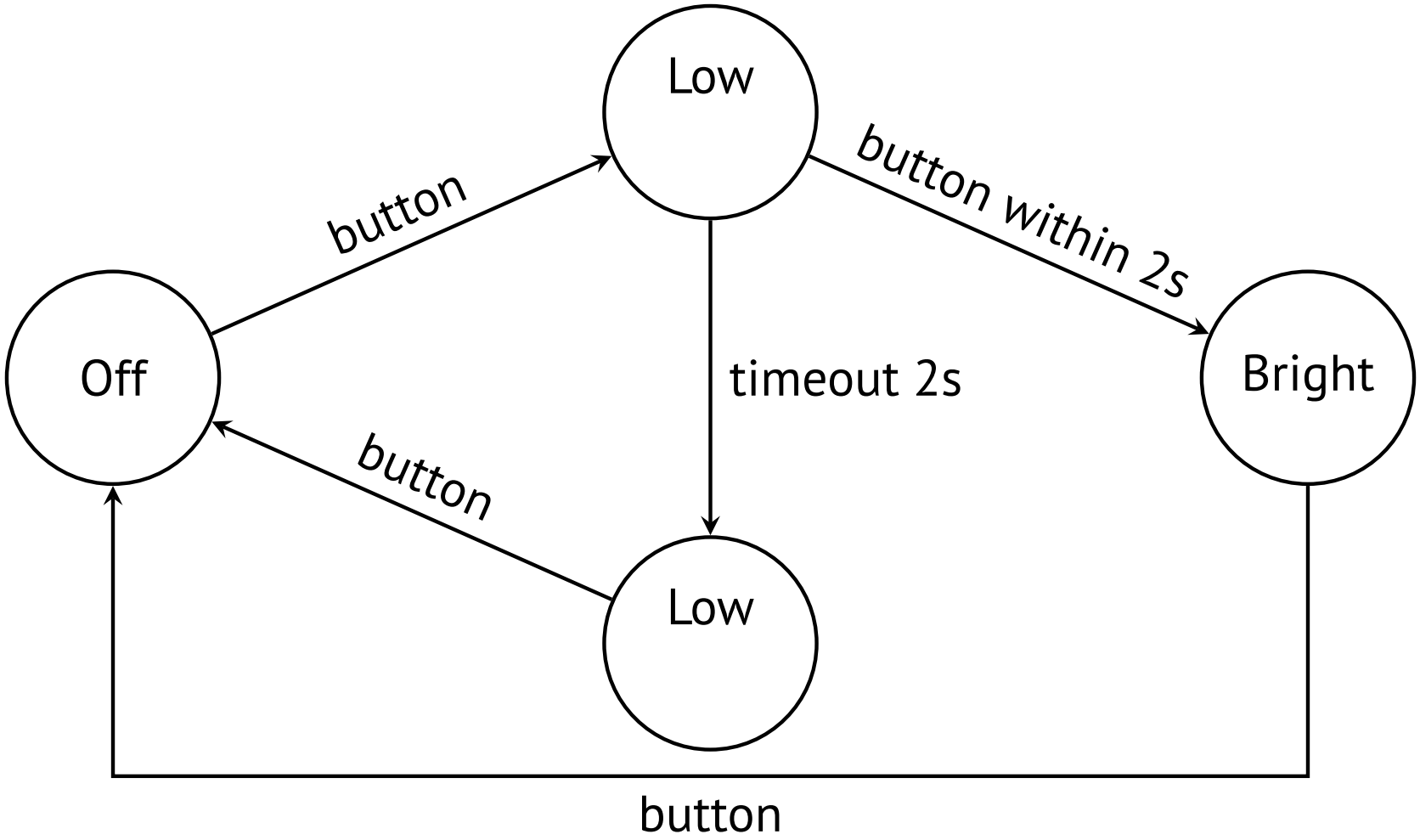
- The lamp can be in three states: off, low light or bright light. If the lamp is off you can turn it on (low light) by pressing a button. If turn you turn the lamp on by pressing the button rapidly two times, within 2s, then it will have a brighter light. If the light is on you turn it off by pressing the button.

A fancy lamp

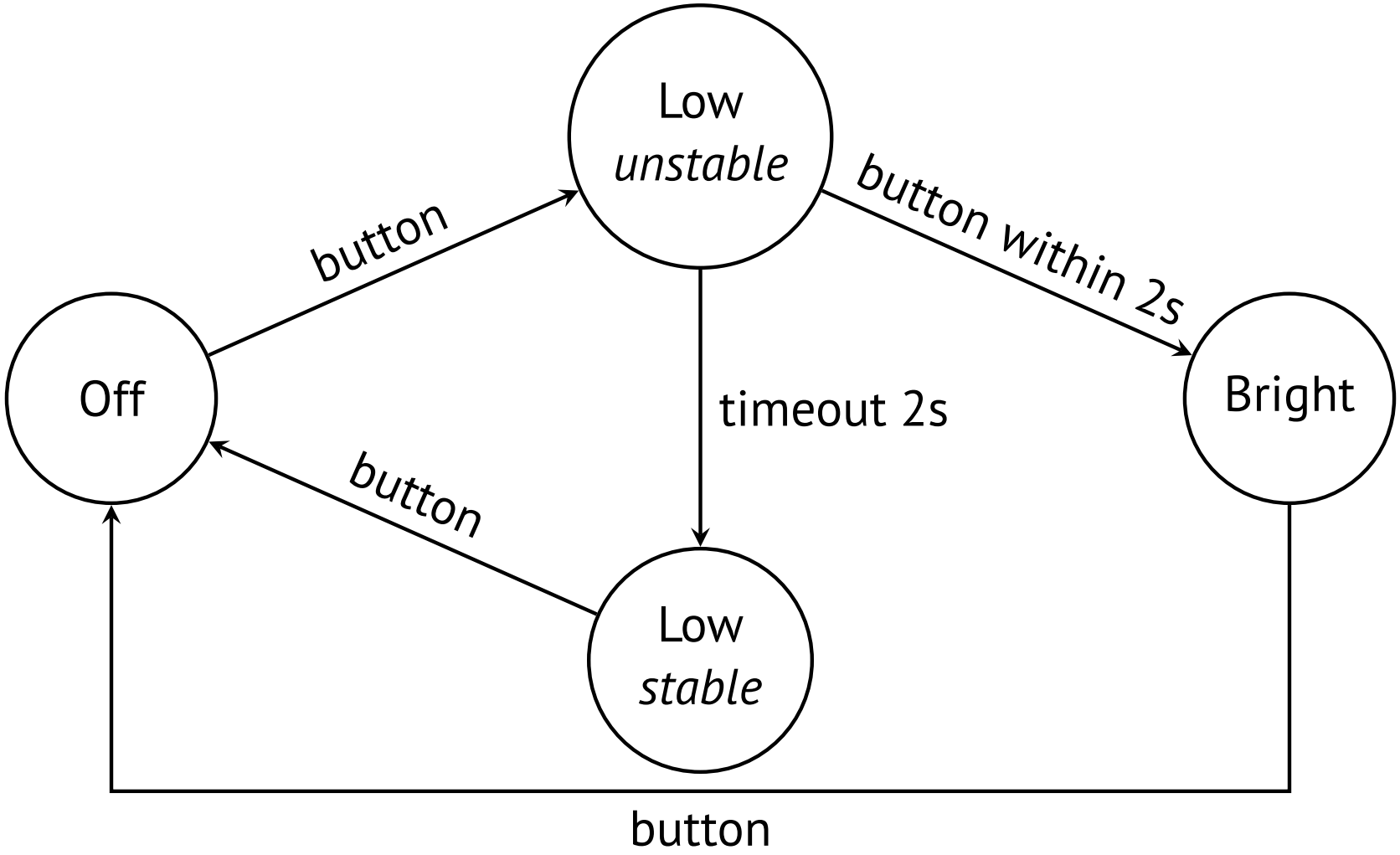
- The lamp can be in three states: off, low light or bright light. If the lamp is off you can turn it on (low light) by pressing a button. If turn you turn the lamp on by pressing the button rapidly two times, within 2s, then it will have a brighter light. If the light is on you turn it off by pressing the button.



Example State Machine: A fancy lamp



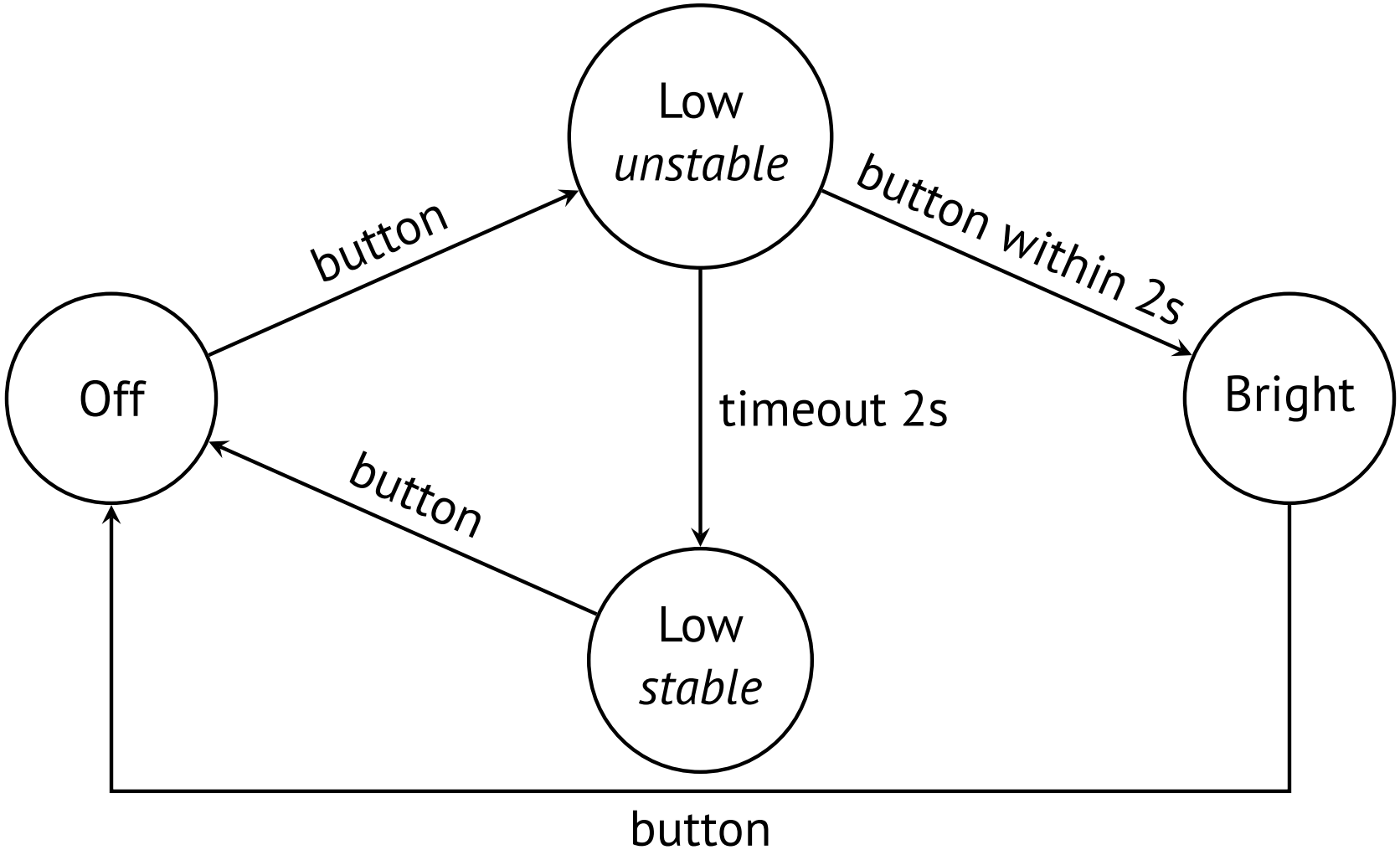
Example State Machine: A fancy lamp



Part III

Implementation with `gen_statem`

Example State Machine: A fancy lamp



Lamp callback module for gen_statem, part 1

```
-module(lamp).

% Public API
-export([start/0, button/1, stop/1]).

% gen_statem callbacks
-export([...]).
-behaviour(gen_statem).

% Public API
start() ->
    gen_statem:start(?MODULE, {}, []).

button(Lamp) ->
    gen_statem:cast(Lamp, button).

stop(Lamp) ->
    gen_statem:stop(Lamp).
```

Lamp callback module for gen_statem, part 2

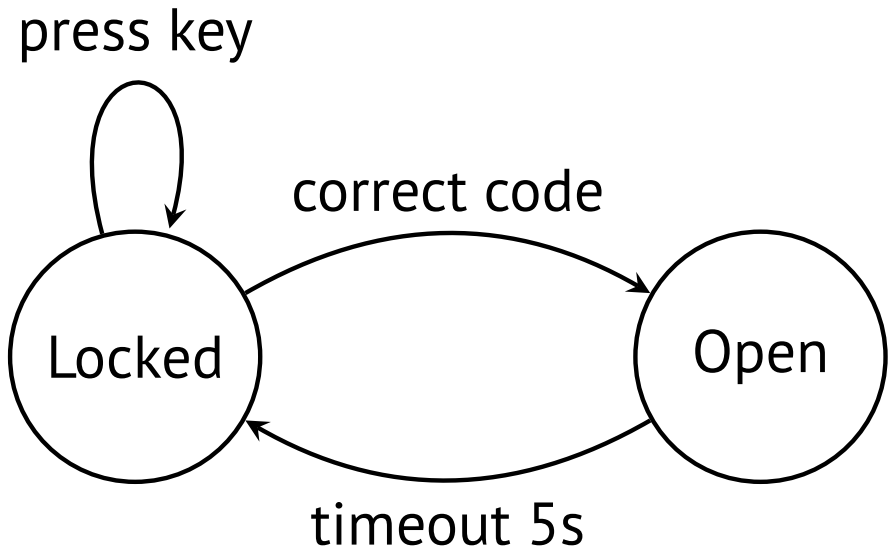
```
callback_mode() -> state_functions.

init({}) -> {ok, off, nothing}.

off(cast, button, Data) ->
    io:format("turn on low light~n"),
    {next_state, low_unstable, Data, 2000}.
low_unstable(cast, button, Data) ->
    io:format("brighten light~n"),
    {next_state, bright, Data};
low_unstable(timeout, _, Data) ->
    io:format("stabilise low light~n"),
    {next_state, low_stable, Data}.
low_stable(cast, button, Data) ->
    io:format("turn off low light~n"),
    {next_state, off, Data}.
bright(cast, button, Data) ->
    io:format("turn off bright light~n"),
    {next_state, off, Data}.
```

Example State Machine: A Door

- A door can be locked or open. To open (unlock) the door you press a code on a keypad. The door automatically locks after 5s.

Example State Machine: A Door

- A door can be locked or open. To open (unlock) the door you press a code on a keypad. The door automatically locks after 5s.



Door callback module for gen_statem, part 1

```
-module(door).
-behaviour(gen_statem).
-export([...]).

start(Code) ->
    gen_statem:start({local, door}, door,
                     lists:reverse(Code), []).

key(Digit) ->
    gen_statem:cast(door, {key, Digit}).

stop() ->
    gen_statem:stop(door).
```

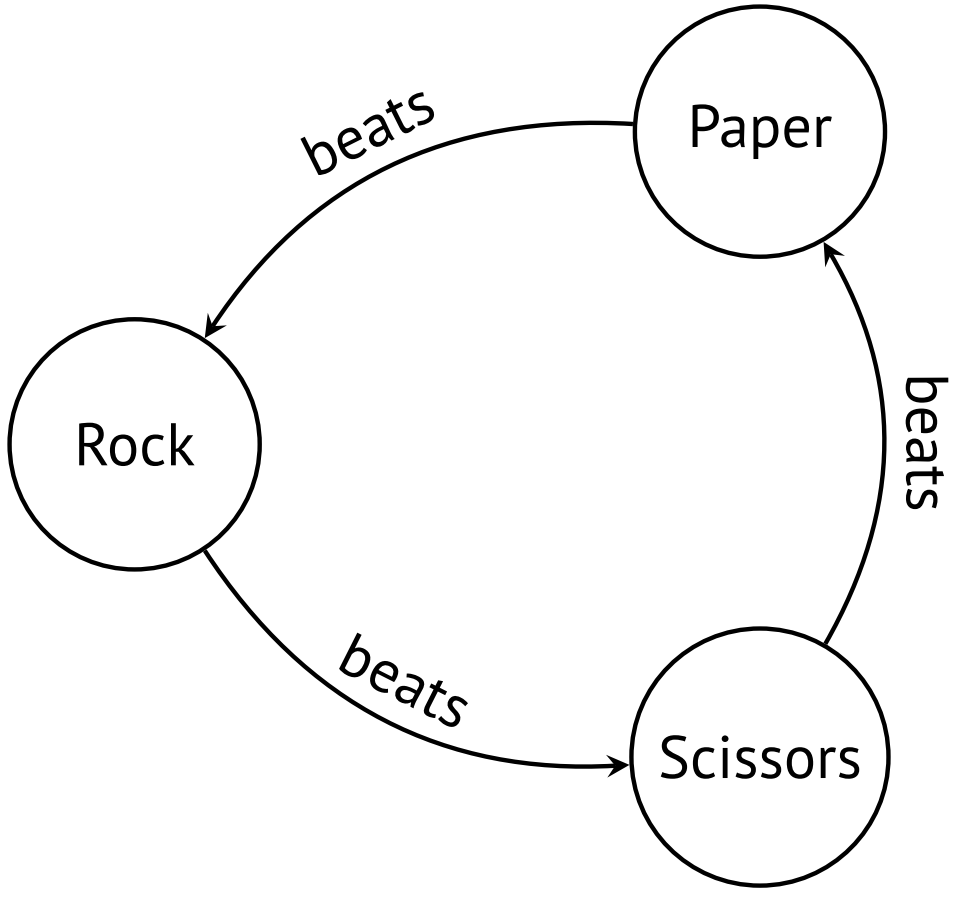
Door callback module for gen_statem, part 2

```
locked(cast, {key, Digit}, {SoFar, Code}) ->
    beep(Digit),
    case [Digit|SoFar] of
        Code ->
            do_unlock(),
            {next_state, open, {[], Code}, 5000};
        Incomplete when length(Incomplete) < length(Code) ->
            {next_state, locked, {Incomplete, Code}};
        _Wrong ->
            thats_not_gonna_do_it(),
            {keep_state, {[], Code}}
    end.

open(timeout, _, State) ->
    do_lock(),
    {next_state, locked, State}.
```

Part IV

Summary

Rock, Paper, Scissors



Rock, Paper, Scissors – Read the Assignment

- Implement a *game server*
- A game server consists of
  - a *game broker*
  - a number of *game coordinators*.

Summary

- 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.
- Use `gen_server` for building robust servers.
- Use `gen_statem` for servers that need to keep track of complex protocols.
- This week's assignment: Rock–Paper–Scissors