# Advanced Programming

## OTP Design Principles

Oleks Shturmov
oleks@oleks.info

with slides adapted from
Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

October, 2021

# Part I

# OTP Design Principles

# OTP Design Principles — Overview

An Erlang application is a collection of processes

Not an OTP Design Principle, but a fact of life

# OTP Design Principles — Overview

An Erlang application is a collection of processes

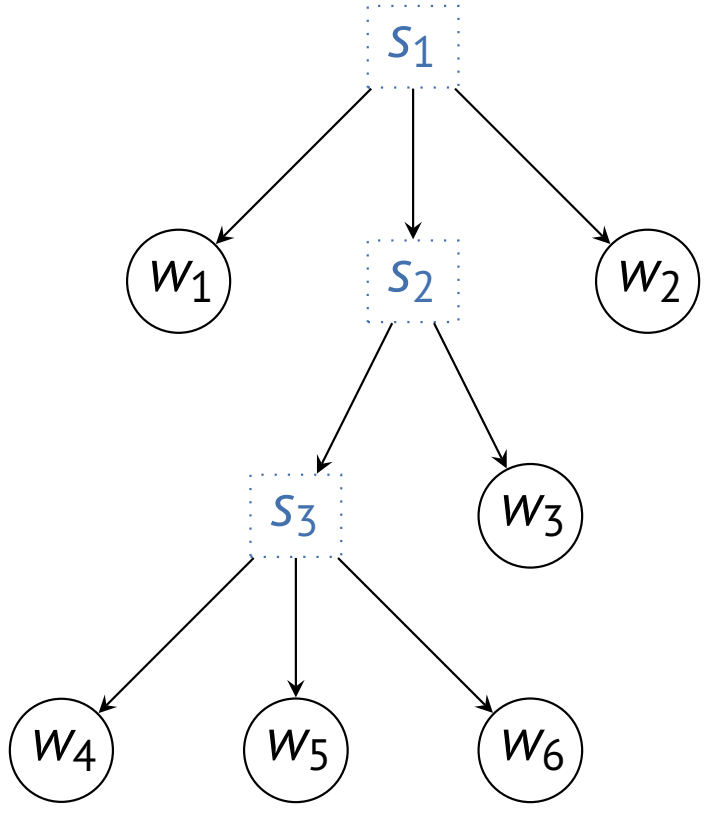
Not an OTP Design Principle, but a fact of life

Now to the Desing Principles:

- Each process MAY BE either a supervisor (□) or a worker (◯)
- A supervisor MAY adhere to a certain **supervisor behaviour**
- Workers MAY adhere to certain **other behaviours**, among them:
  - A **gen**eric **server**
  - A **gen**eric **state m**achine
  - A **gen**eric **event** handler

# Supervisor Behaviour (1/2)

- Each process MAY BE either a supervisor (□) or a worker (◯)
  - The job of a supervisor is to orchestrate workers…
  - …or other supervisors, forming a **supervision tree**:



# Supervisor Behaviour (2/2)

- Specify a **restart strategy**
  - If a child fails, restart *that* child
  - If a child fails, restart *all* children
  - …

# Supervisor Behaviour (2/2)

- Specify a **restart strategy**
  - If a child fails, restart *that* child
  - If a child fails, restart *all* children
  - …
- Specify the **maximum restart intensity**
  - Number of restarts that may take place over a period of time

# Supervisor Behaviour (2/2)

- Specify a **restart strategy**
  - If a child fails, restart *that* child
  - If a child fails, restart *all* children
  - …
- Specify the **maximum restart intensity**
  - Number of restarts that may take place over a period of time
- Instruct supervisor to terminate **if significant children fail**

# Supervisor Behaviour (2/2)

- Specify a **restart strategy**
  - If a child fails, restart *that* child
  - If a child fails, restart *all* children
  - …
- Specify the **maximum restart intensity**
  - Number of restarts that may take place over a period of time
- Instruct supervisor to terminate **if significant children fail**
- Last, but not least, **specify the children!**

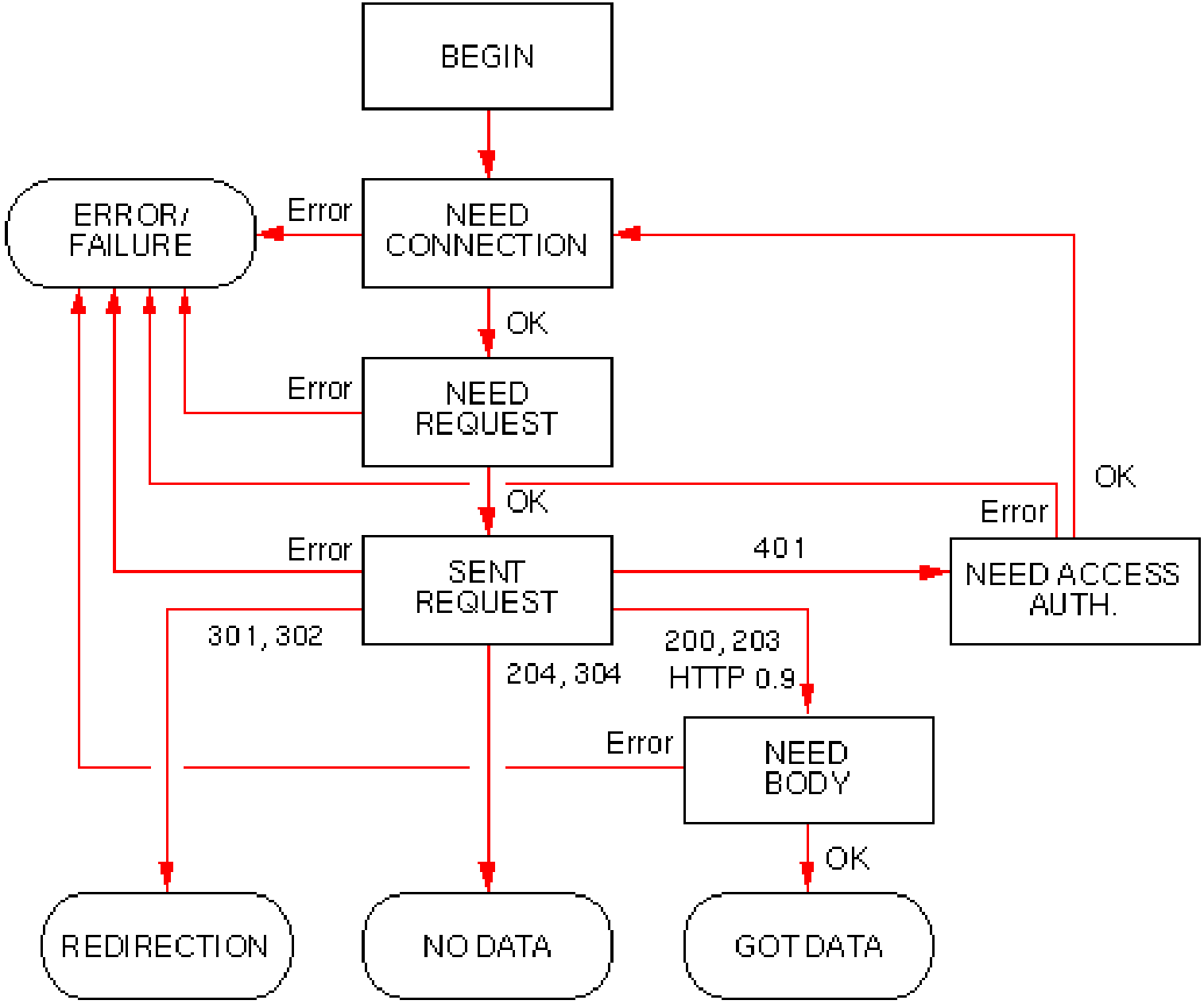
# `gen_server` — generic server

- MAY be useful whenever you have a process that needs to act as a server to some clients
- Eliminates the need to declare a loop function
- Eliminates the need to explicitly deal with communication
  - Use `gen_server:cast/2` for **async**hronous communication
  - Use `gen_server:call/2` for **sync**hronous communication
  - **NB!** Don't mix `!` and `receive` with `gen_server`
- Other freebies:
  - Processes are registered; no need to carry a `Pid` around
  - Working with supervisors
  - Hot-swapping code

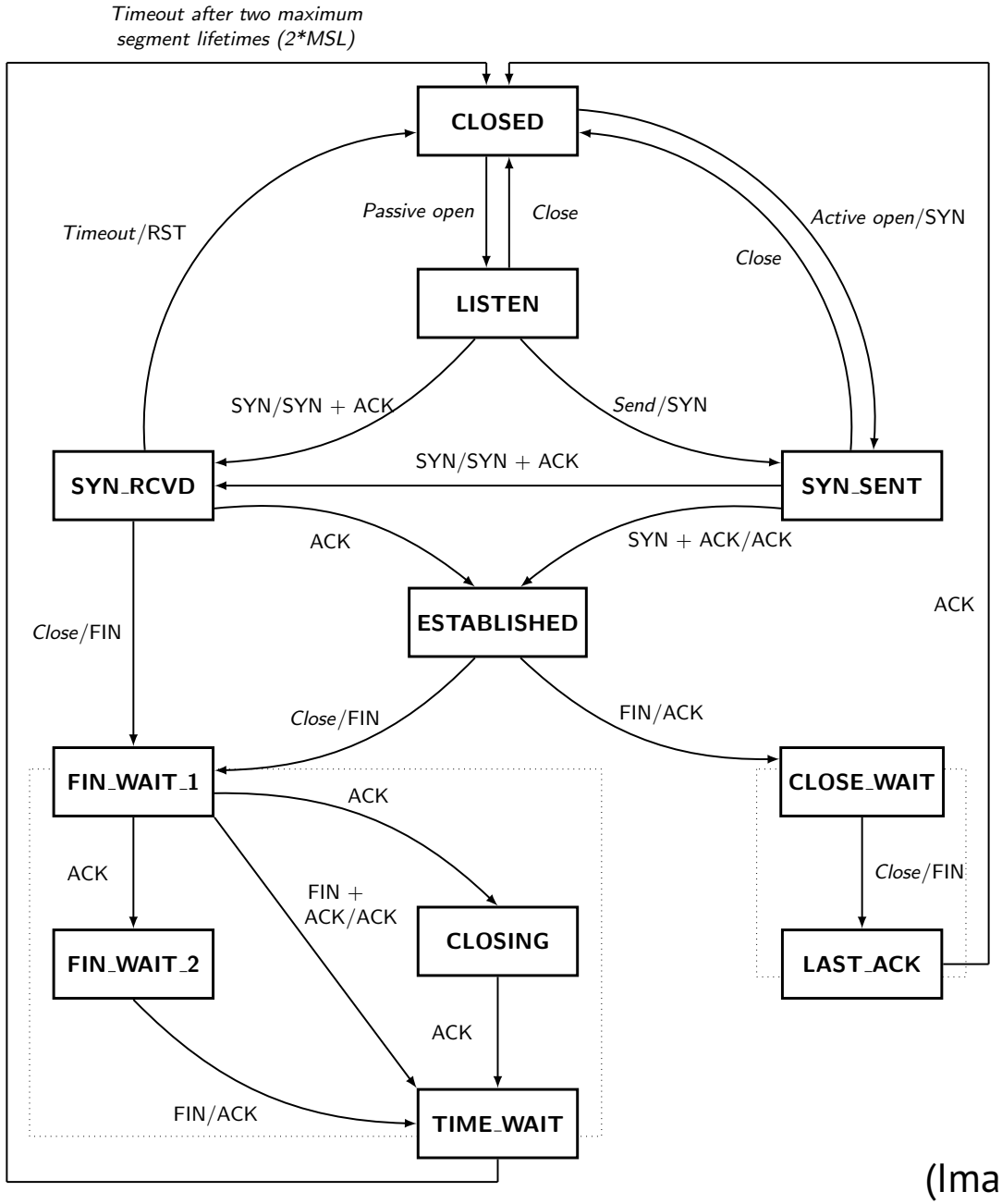
# `gen_statem` — generic state machine

- *MAY* be useful whenever you can see that you can model (i.e., draw) your module as a state machine…with data…so, always?
- A `gen_server` also has state, but with `gen_statem`, we can make states and state transitions explicit
- Has **fine-grained timeout control**—useful when you would like to make sure events and state transitions happen in a timely manner
- Carries over many of the goodies from `gen_server`, in particular, we also have `gen_statem:call/2` and `gen_statem:cast/2`

# HTTP Client State Machine



# TCP (RFC 793) State Machine



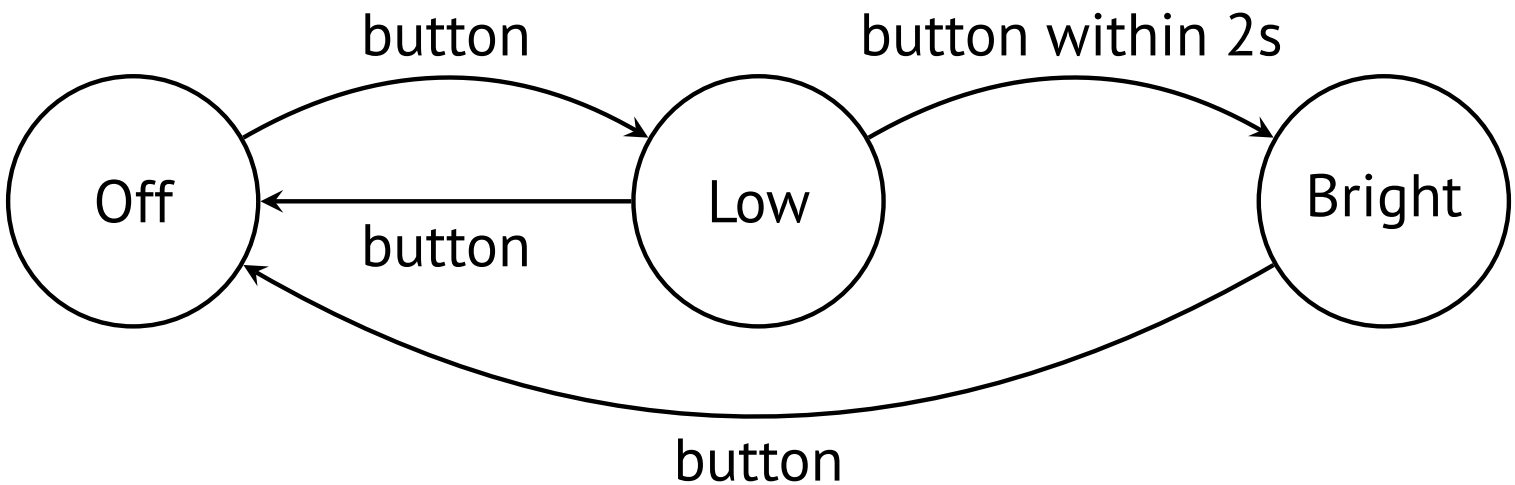

(Image credit Ivan Griffin)

# A fancy lamp

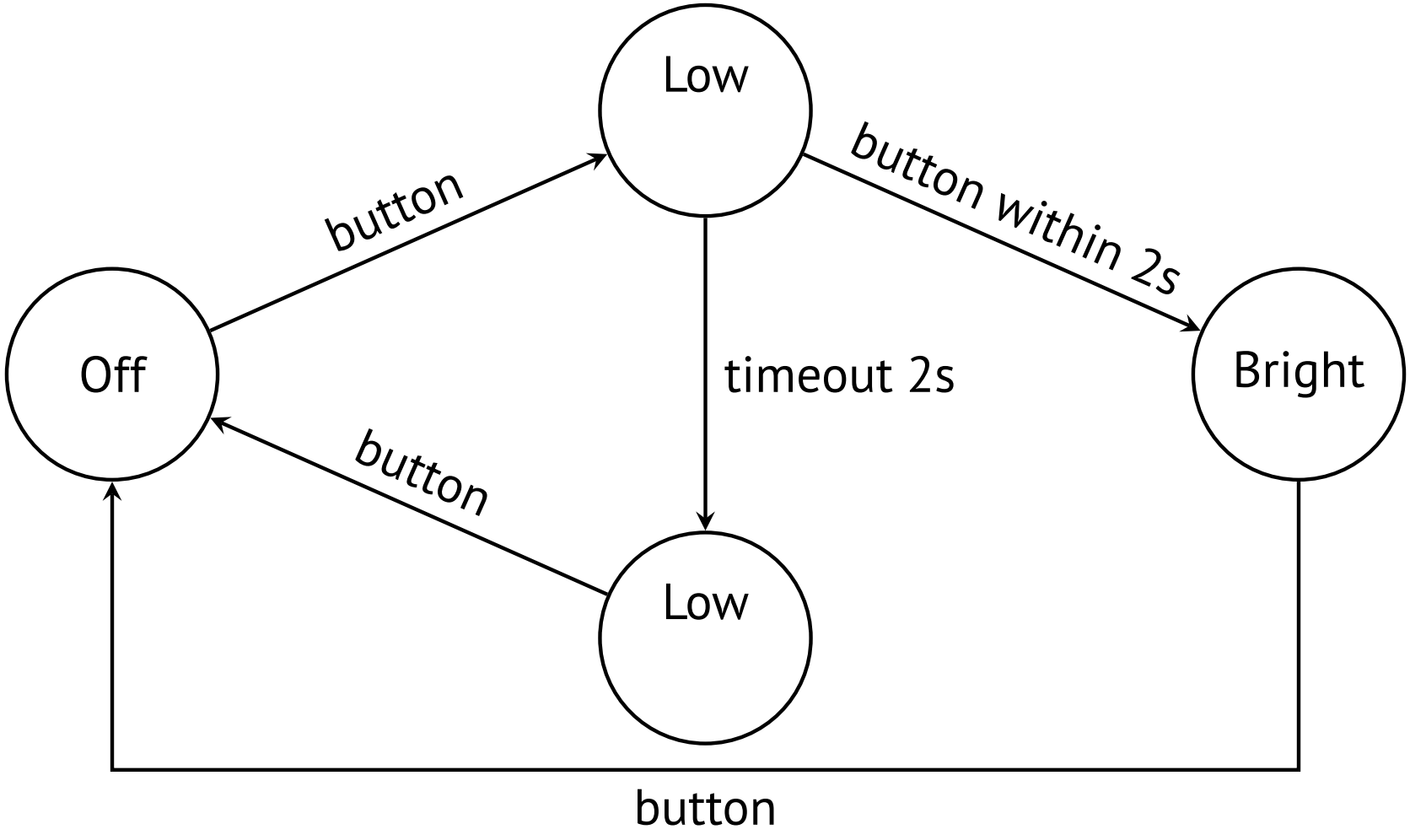
- The lamp can be in three states: off, low light or bright light. If the lamp is off you can turn it on (low light) by pressing a button. If turn you turn the lamp on by pressing the button rapidly two times, within 2s, then it will have a brighter light. If the light is on you turn it off by pressing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

# A fancy lamp

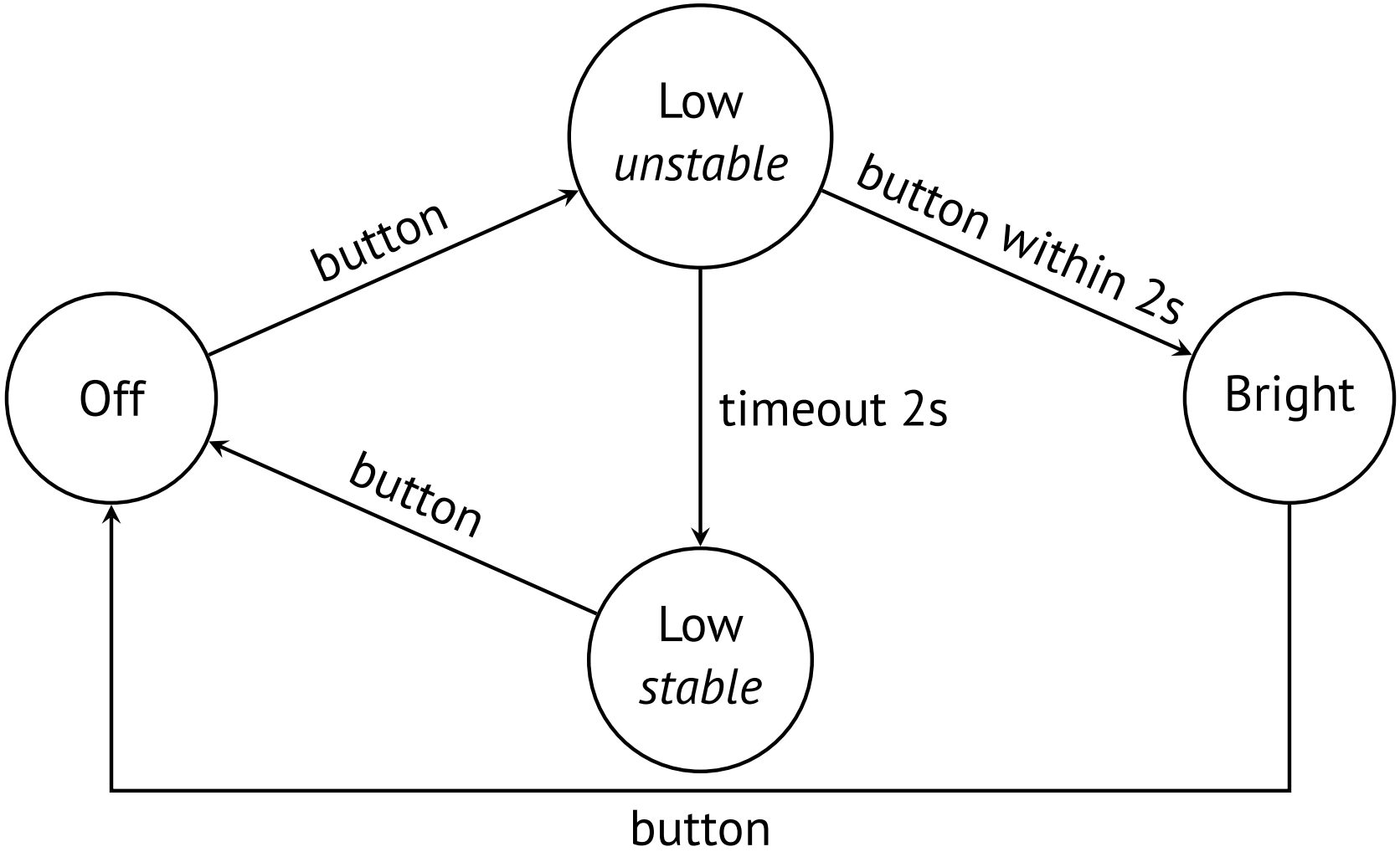
- The lamp can be in three states: off, low light or bright light. If the lamp is off you can turn it on (low light) by pressing a button. If turn you turn the lamp on by pressing the button rapidly two times, within 2s, then it will have a brighter light. If the light is on you turn it off by pressing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



# Example State Machine: A fancy lamp



# Example State Machine: A fancy lamp



# 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

# Callback Mode

`state_functions`

States are represented by module functions with signatures of the form

```
Module:StateName(EventType, EventContent, Data)
```

where `Data` bears additional state data

`handle_event_function`

States are represented as Erlang terms, and there is a single handler function in the modeule, having the form

```
Module:handle_event(
  EventType, EventContent, State, Data)
```

# Event Types

`cast`
: An asynchronous event

`{call, From}`
: A synchronous event—must be replied to with `gen_statem:reply(From, Reply)`

`timeout`
: An event timeout — the state machine has been waiting for too long for an event to occur

`state_timeout`
: The state machine has been waiting for too long for the state to change

`{timeout, Name}`
: The statate machine has been waiting for too long (e.g., for a sequence of events or state changes to occur)

`internal`
: For the state machine to issue events on its own

# Staate Callback Return Value

- `{next_state, NextState, NewData}`
- `{next_state, NextState, NewData, Actions}`
- `{keep_state, NewData}`
- `{keep_state, NewData, Actions}`
- ...

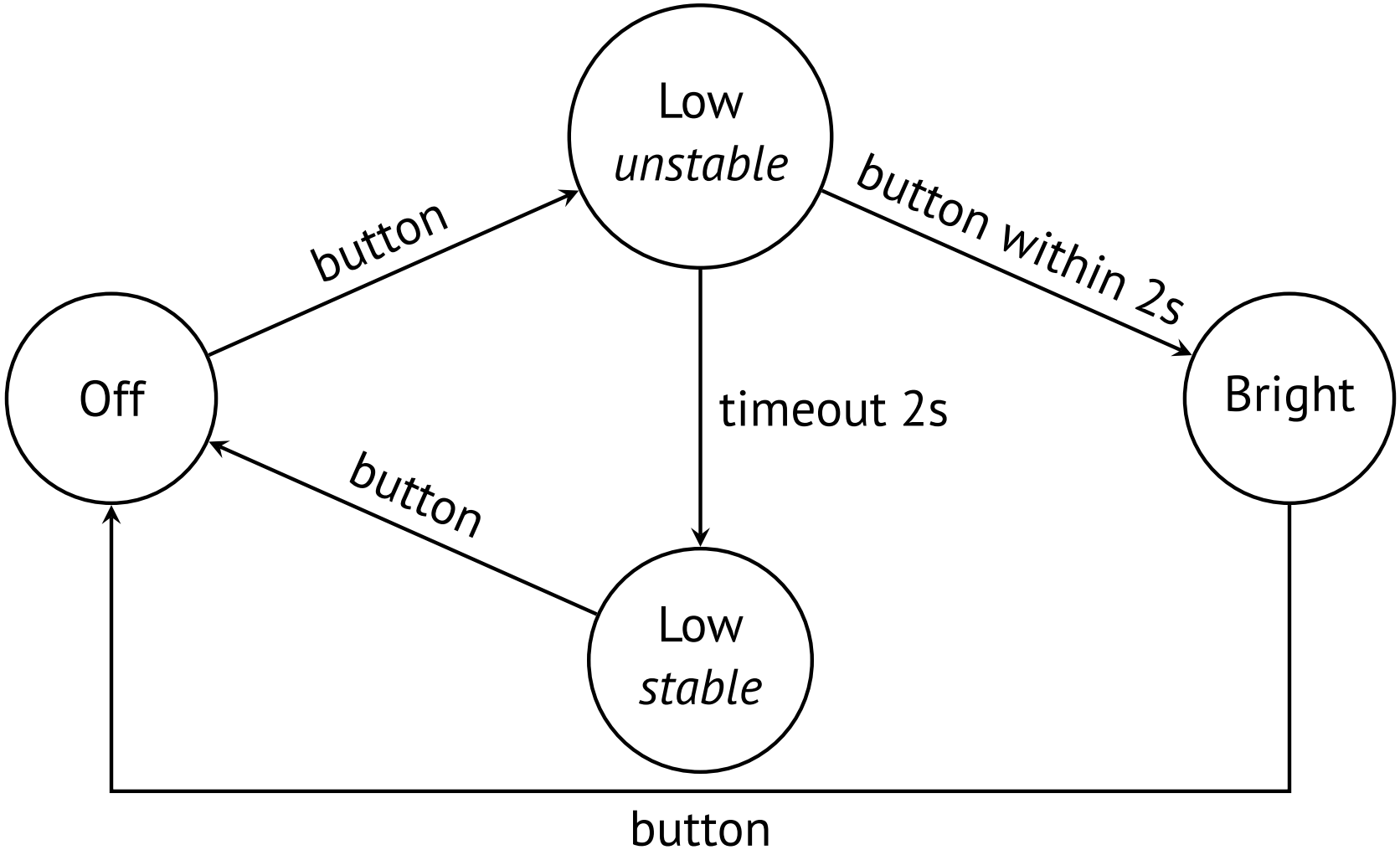
# Transition Actions

- `postpone`
- `{state_timeout, Time, EventContent}`
- `{reply, From, Reply}`
- `...`

# Part II

## Implementation with `gen_statem`

# Example State Machine: A fancy lamp



# Lamp callback module for gen_statem, part 1

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

start() -> gen_statem:start({local, lamp}, ?MODULE,[],[]).
button() -> gen_statem:cast(lamp, button).
stop() -> gen_statem:stop(lamp).

callback_mode() -> state_functions.
init(_Code) ->
    lamp_off(),
    {ok, off, nothing}.
terminate(normal, _StateName, _StateData) -> ok.
code_change(_Vsn, State, Data, _Extra) -> {ok,State,Data}.
```

# Lamp callback module for gen_statem, part 2

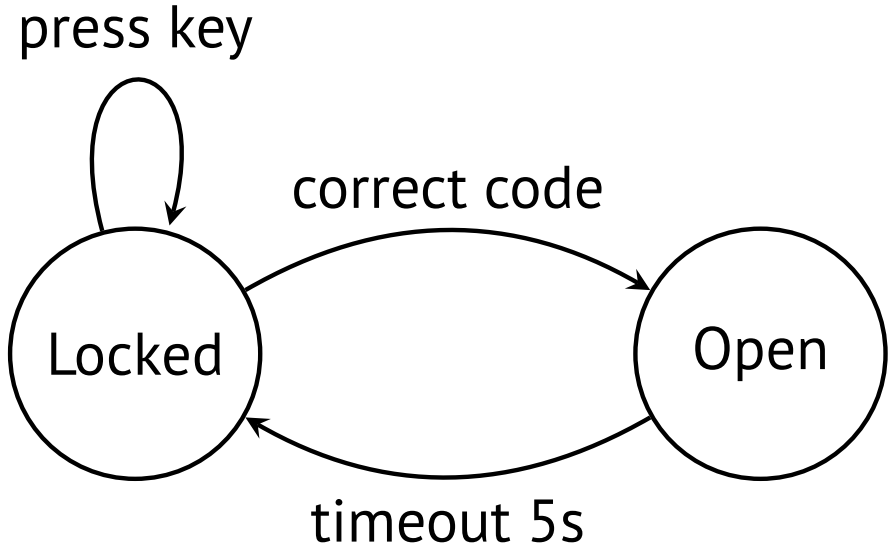
```
off(cast, button, Data) ->
    lamp_low_light(),
    {next_state, low_unstable, Data, 2000}. % timeout after 2s

low_unstable(cast, button, Data) ->
    lamp_bright_light(),
    {next_state, bright, Data};
low_unstable(timeout, _, Data) ->
    {next_state, low_stable, Data}.

low_stable(cast, button, Data) ->
    lamp_off(),
    {next_state, off, Data}.

bright(cast, button, Data) ->
    lamp_off(),
    {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 lock 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 lock 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).
```

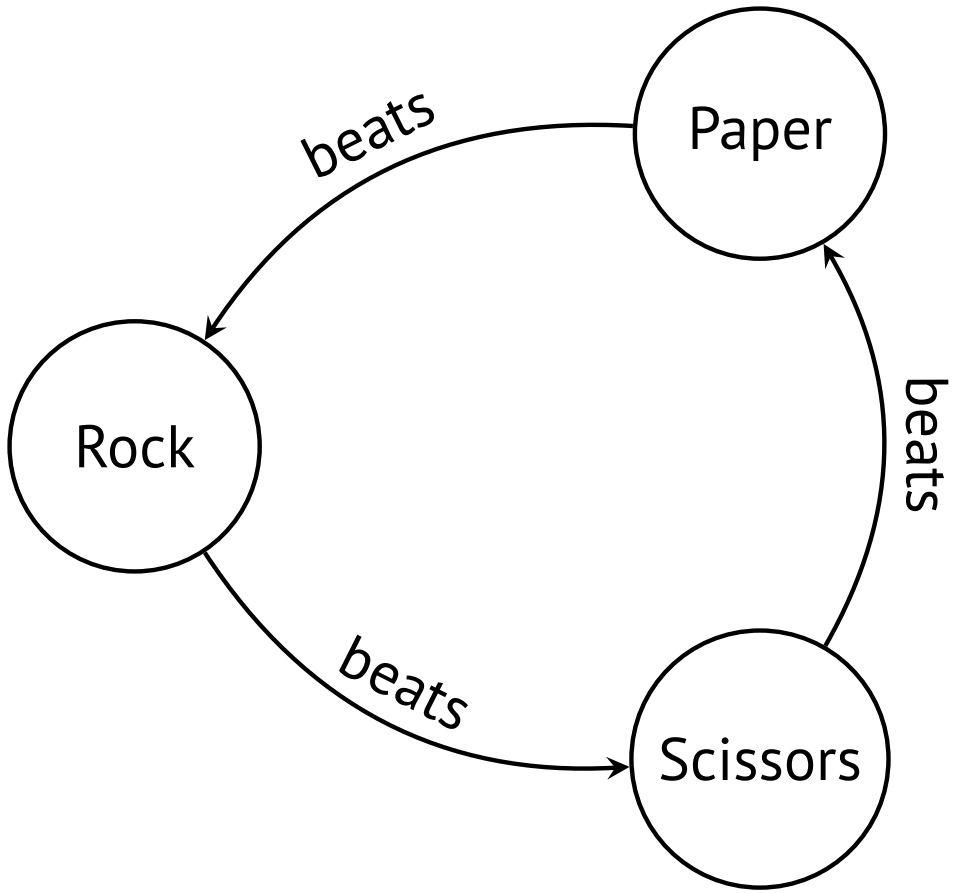
# Door callback module for gen_statem, part 2

```
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 III

## 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