Advanced Programming

Riding the OTP

Ken Friis Larsen
`kflarsen@diku.dk`

Department of Computer Science
University of Copenhagen

October 12, 2017

Today’s Menu

- Library code for making robust servers
- Open Telecom Platform (OTP)

Generic Servers

- Goal: Abstract out the difficult handling of concurrency to a generic library
- The difficult parts:
    - The `start–blocking(/async)–loop` pattern
    - Registering processes
    - Supervisors
    - Hot-swapping of code

Basic Server Library

```
start(Name, Mod) ->
  register(Name, spawn(fun() -> loop(Name, Mod, Mod:init())
                       end)).
blocking(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

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

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

Hot Code Swapping

```
swap_code(Name, Mod) -> blocking(Name, {swap_code, Mod}).
blocking(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.
```

What is the behaviour

- For a callback module to work with `basicserver` and `codeswap` it need to export two functions `init` and `handle`.
  It would be great if someone could help us get that right...
- The compiler can help us
- Add the following to `basicserver.erl`:

```
-callback init() -> State :: term().
-callback handle(Arg :: term(), State :: term()) ->
    { ok, State :: term() } |
    { {error, Reason :: term()}, State :: term() }.
```

- Add the following to `pb.erl`:

```
-behaviour(basicserver).
```

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 write them by need.)

From the Original Kaboose!

- The `guess` function should only be used when there is an active question being asked.
- Players can only `join`, `leave` and `rejoin` when there is no active question.
- New questions can only be added while there are no players in the room.
- A room can be locked (and later unlocked) so no players can join the room
- The room should be locked with a code, the last process that unlock the room is the conductor

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

(you can do it by need.)

Callback module for gen_statem, part 1

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

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

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

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

Callback module for gen_statem, part 2

```
locked(cast, {button, 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}.
```

Flamingo or Revision

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` (or `gen_fsm`) for servers that can be in different states.
- This week's assignment: Flamingo
- The TAs cannot spend a week without you. So there is exercise labs in rooms 1-0-30, 1-0-34 and 1-0-37 at UP1 Tuesday, Oct 17 13:00–

Exam

- One week take-home project (3/11–10/11)
- Hand in via Digital Exam
- Max group size is **1** (one)
- The University has a zero-tolerance policy against exam fraud (including assisting).