Advanced Programming

Erlang OTP

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

October, 2021

Part I

Pre-lecture – Separation of Concerns

Recap – Stateful Server

- Organise your code in modules
- Functions are pure (stateless).
- Processes can be used as the guardians of state.
- We organise our code as micro-servers that manage some data (a.k.a. state) that can be manipulated via a client API (a.k.a concurrent objects).
- Functions starts processes, processes runs functions, functions are defined in modules.

Separation of Concerns – API

```
-export([start/0, add_item/3, all_items/1, finish/2]).

start()                -> spawn(fun () -> loop(init()) end).
add_item(TL, Desc, Due) -> request_reply(TL, {add, {Desc, Due}}).
all_items(TL)          -> request_reply(TL, all_items).
finish(TL, Index)      -> request_reply(TL, {finish, Index}).
```

Separation of Concerns – Data Manipulation

```
init() -> [].

handle_call({add, {Description, Due}}, Items) ->
    Item = #{ description => Description,
              due => Due},
    {ok, [Item | Items]};
handle_call(all_items, Items) ->
    {{ok, Items}, Items};
handle_call({finish, Index}, Items) ->
    try {Before, [_Idx | After]} = lists:split(Index-1, Items),
        {ok, Before ++ After}
    catch
        error : _ -> {{error, index_out_of_bounds}, Items}
    end.
```

Separation of Concerns – Communication

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

loop(Data) ->
  receive
    {From, Request} ->
      {Res, NewData} = handle_call(Request, Data),
      From ! {self(), Res},
      loop(NewData)
  end.
```

TODO Usage

```
2> TL = todo_sc:start().
<0.86.0>
3> todo_sc:all_items(TL).
{ok,[]}
4> L = [{"make slides for erlang/OTP lecture", {2021,10,10}},
4>      {"OnlineTA for assignment 5", {2021,10,11}}].
5> lists:map(fun({Desc, Due})->todo_sc:add_item(TL, Desc, Due)
5>           end, L).
[ok,ok]
6> todo_sc:all_items(TL).
{ok,[#{description => "OnlineTA for assignment 5",
       due => {2021,10,11}},
     #{description => "make slides for erlang/OTP lecture",
       due => {2021,10,10}}]}
7> todo_sc:finish(TL, 2).
ok
```

Using Multiple Protocols

- The `add_item` function can never fail, no need to block and wait for the reply
- Change the API function:

```
add_item(TL, Desc, Due) -> nonblock(TL, {add, {Desc, Due}}).
```

- Update the communication part:

```
nonblock(Pid, Request) -> Pid ! {self(), Request}.

loop(Data) ->
  receive
    {From, Request} ->
      case handle_call(Request, Data) of
        {reply, Res, NewData} ->
          From ! {self(), Res},
          loop(NewData);
        {noreply, NewData} ->
          loop(NewData)
      end
  end.
```

Using Multiple Protocols – Part 2

- Update data-manipulation:

```
handle_call({add, {Description, Due}}, Items) ->
  Item = #{ description => Description,
            due => Due},
  {noreply, [Item | Items]};
handle_call(all_items, Items) ->
  {reply, {ok, Items}, Items};
handle_call({finish, Index}, Items) ->
  try {Before, [_Idx | After]} = lists:split(Index-1,Items),
      {reply, ok, Before ++ After}
  catch
      error:_ -> {reply, {error,index_out_of_bounds},Items}
  end.
```

Using Multiple Protocols – Alternative, Communication

```
nonblock(Pid, Request) -> Pid ! {cast, self(), Request}.
request_reply(Pid, Request) ->
  Pid ! {call, self(), Request},
  receive {Pid, Response} -> Response
  end.

loop(Data) ->
  receive
    {call, From, Request} ->
      case handle_call(Request, Data) of
        {reply, Res, NewData} ->
            From ! {self(), Res},
            loop(NewData);
        {noreply, NewData} -> loop(NewData)
      end;
    {cast, _From, Req} -> NewData = handle_cast(Req, Data),
                          loop(NewData)
  end.
```

Using Multiple Protocols – Alternative, Data Manipulation

```
handle_call(all_items, Items) ->
  {reply, {ok, Items}, Items};
handle_call({finish, Index}, Items) ->
  try {Before, [_Idx | After]} = lists:split(Index-1, Items),
      {reply, ok, Before ++ After}
  catch
    error : _ -> {reply, {error, index_out_of_bounds}, Items}
  end.

handle_cast({add, {Description, Due}}, Items) ->
  Item = #{ description => Description,
            due => Due},
  [Item | Items].
```

Today's Menu

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

Part II

Generic Servers

Generic Servers

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

Simple Server Library

```
start(Mod) -> spawn(fun() -> loop(Mod, Mod:initialise()) end).

request_reply(Pid, Request) -> Pid ! {call, self(), Request},
                               receive {Pid, Reply} -> Reply end.
nonblocking(Pid, Request) -> Pid ! {cast, Request}.

loop(Mod, Data) ->
  receive
    {call, From, Request} ->
      case Mod:handle_call(Request, Data) of
        {reply, Res, NewData} ->
          From ! {self(), Res},
          loop(NewData);
        {noreply, NewData} ->
          loop(NewData)
      end;
    {cast,  Request} ->
      NewData = handle_cast(Request, Data),
      loop(NewData)
  end.
```

Behaviour

```
-callback initialise() -> State :: term().
-callback handle_call(Arg :: term(), State :: term()) ->
    { reply, Response:: term(), State :: term() } |
    { noreply, State :: term() }.
-callback handle_cast(Arg :: term(), State :: term()) ->
    { State :: term() }.
```

Register Server Library

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

request_reply(PidOrName, Request) ->
  PidOrName ! {self(), Request},
  receive
    {PidOrName, Reply} -> Reply
  end.

loop(Name, Mod, State) ->
  receive
      {From, Request} ->
          {Reply, State1} = Mod:handle(Request, State),
          From ! {Name, Reply},
          loop(Name, Mod, State1)
  end.
```

Behaviour

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

Example: Todo-List Callback Module, 1

```
-module(todo).
-export([start/0, add_item/2, all_items/0, finish/1]).
-export([init/0, handle/2]).
-import(registerserver, [request_reply/2]).

%% Interface
start()           -> registerserver:start(todo, todo).
add_item(Desc, D) -> request_reply(todo, {add, {Desc, D}}).
all_items()       -> request_reply(todo, all_items).
finish(Index)     -> request_reply(todo, {finish, Index}).
```

Example: Todo-List Callback Module, 2

```
%% Callback functions
init() -> [].

handle({add, {Description, Due}}, Items) ->
    Item = #{ description => Description,
              due => Due},
    {ok, [Item | Items]};
handle(all_items, Items) ->
    {{ok, Items}, Items};
handle({finish, Index}, Items) ->
    try {Before, [_Idx | After]} = lists:split(Index-1, Items),
        {ok, Before ++ After}
    catch
        error : _ -> {{error, index_out_of_bounds}, Items}
    end.
```

Todo-lists are important

Suppose that we really most have a todo-list server running at all times.
Using rigorous testing we have a library without any bugs(?!?!).
However, we fear that we'll discover that we really, really, really want some new functionality. What to do?

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.
```

Part III

Open Telecom Platform (OTP)

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.)

Part IV

Summary

Modularity

- Erlang offer different tools for modularity:
  - Functions
  - Modules
  - Processes
  - (Nodes, network, …)
- Be careful when crossing *trust boundaries*.
- Identify and document *assumptions*.

Summary

- Structure your code into the infrastructure parts and the functional parts.
- Use `gen_server` for building robust servers.

Exam

- One week take-home project (5/11–12/11)
- Hand in via Digital Exam
- Check with OnlineTA before submission
- Max group size is **1** (one)
- (Please remember that the University have zero-tolerance policy regarding exam fraud)