Advanced Programming

Exam

Per Steffen Czolbe

Handed in: October 15, 2017

Contents

1 Assignment 1: Assignment Title **1**
1.1 Setup . . . 1
Language and Frameworks . . . 2
Files . . . 2
Program Execution . . . 2
Test Execution . . . 2
1.2 Implementation . . . 3
Assumptions . . . 3
Keeping Things Organized . . . 3
Processes and Communication . . . 3
Guessing While no Question is Active . . . 3
Suggested Improvements to the Kaboose API . . . 4
1.3 Assessment . . . 5
Scope of Test Cases . . . 5
Correctness of Solution . . . 5
Summary of Code Quality . . . 6

Appendices **6**

A Assignment 1 **6**
A.1 actionModuleServer.erl . . . 6
A.2 simpleSupervisor.erl . . . 9

1 Assignment 1: Assignment Title

Description.
The goal of the assignment is to implement the backend part of a Kahoot!-like service called Kaboose!. That is, a Kaboose! server is a central server where you create new quiz rooms. When you have created a quiz you need to add some questions to it before you can play the quiz. When your quiz is ready, you can open the quiz room for players. The players need to join the quiz before they can make guesses. To play a quiz there needs to be exactly one conductor and a number of players.

The play of a quiz is that the conductor makes the questions active in turn, and then the players' guesses are collected. After each question, each player is awarded points based on the correctness of the answer, and the swiftness of reply. [Larsen, 2017]

1.1 Setup

This section gives a brief overview about the structure of the solution and how to execute the code.

Language and Frameworks

Kahoot is implemented in Erlang, a language that handles the concurrency of different quiz rooms very well. A modified version of the basic server library supplied in the lecture is used for communication, and EUnit is used for unit tests. Emake is used to build the project.

Files

The code is organised in multiple files and directories:

- `src/`: the project directory.
  - `kaboose.erl`: the kaboose module.
  - `room.erl`: a module containing the implementation of a game room.
  - `activeRoom.erl`: a module containing the implementation of an active game room.
  - `test/`: a directory with test cases.
    - `kaboose_tests.erl`: the tests for the kaboose module.
  - `Emakefile`: a file congaing Emake configuration.
  - `ebin/`: a directory containing the binary files generated by Emake.

Program Execution

To compile and load the kaboose module navigate to the project directory and run:

```
$ erl
1> make:all([load]).
```

This triggers Emake to compile and load all files in the project. Note that a `src/ebin/` directory must exist.

Now functions of the kaboose module can be called from the eshell.

Test Execution

First, follow the steps in the previous section to compile and load all the project files. Afterwards, all test cases can be run in verbose mode with a single command:

```
2> eunit:test(kaboose, [verbose]).
```

1.2 Implementation

This section contains a description of the implementation. First, our assumptions regarding the assignment are stated. Next, concepts of how to tackle the harder parts of the implementation are discussed.

Assumptions

Erlang is a dynamically typed language. This allows for a wide range of possible errors as a result of arbitrarily structured inputs. Instead of performing a sanity-check on each input, we apply the ”perfect-world” assumption: a user always supplies inputs in the expected format.

Keeping Things Organized

To structure the code, we first identify three different layers of the kaboose application:

- kaboose, the main server managing the creation of game rooms.
- room, a type of process that manages adding questions to a room and creating active rooms.
- activeRoom, a type of process that resembles an active game.

Each of these three layers represents a different type of process. To structure the source code, each of them is implemented in a separate module and file. To still conform with the API as defined by the assignment, the kaboose module offers functions for the other modules as well, but merely calls the implementation of those functions in the other modules.

Processes and Communication

To facilitate the communication and basic functionality of processes, we use a modified version of the `basic_server` module presented in the lecture. To suit our needs, we performed two modifications to it:

- The initial implementation registered processes as names. But as registered names have to be unique, a trivial implementation would only allow one process of each type to run at a time. To circumvent this restriction without adding too much complexity, we decided to not register processes and instead just use the pid to refer to processes.
- The basic_server was only offering a request-reply communication pattern. For this solution, a asynchronous communication pattern was added.
- Kill function has been added so a process can terminate itself.

With two different communication pattern available, we choose which pattern to use for each function based on if a result needs to be communicated. This used patterns are shown in table 1.

Guessing While no Question is Active

A player of a kaboose game can potentiality guess the answer of a question at any time, even when there is no question currently asked.

We decided to ignore these guesses, the player is neither awarded points for these guesses, nor is he punished. However, the messages received by the activeRoom process still have to be handeled. Not handling them would mean they remain the the message queue and potentially get interpreted as answers to the next question. Consequently, this solution handles incoming guesses as soon as they arrive, reads from it’s internal state whether there is currently an active question, and ignores the guess if there is not.

Table 1: Communication patterns used in API functions

function	communication pattern
get_a_room	request-reply
add_question	request-reply
get_questions	request-reply
play	request-reply
next	request-reply
timesup	request-reply
join	request-reply
leave	async
rejoin	async
active room notifying the conductor about players joining/leaving	async

Suggested Improvements to the Kaboose API

The kaboose API could be improved by:

1. Possible implementations of the opaque data type `Cref` are severely limited by the signature of the `next()` function having `Cref` as an input.
2. Many error cases are not specified in the assignment. This makes it hard to determine what should be handled properly, and which cases should lead to a runtime error.

1.3 Assessment

This section contains the assessment of the presented solution.

Scope of Test Cases

The `kaboose_tests.erl` file contains 22 tests, that cover all the Kaboose functionality. Tests are verifying both positive and negative cases, e.g. verifying if no question will be added in case of empty answer list. In the report, only more interesting parts of the code will be described.

- It was necessary to test if only Conductor can use `next`, `timesup`. To do that in the test a method that simulates another pid was created:

```
fakeConductorNext(Aroom, Pid) ->
Pid ! kaboose:next(Aroom).
```

Than it was possible to verify if the caller is actual conductor:

```
receive
        Message -> Message
end,
?assertMatch({error, who_are_you}, Message).
```

- Additionally it was also verified if messages about player status are received only be the conductor. Data should not be leaked.

```
kaboose:join(Aroom, "Konrad"),
receive
Message -> Message
end,
Me = self(),
?assertMatch({Me,{player_joined, "Konrad", _}}, Message).
```

- Another important and tricky part was checking if points are granted with the respect to the time of answers being received by the server:

```
kaboose:next(Aroom),
timer:sleep(600),
kaboose:guess(Aroom, Ref, 1),
{ok,[1,_],Dict,_,true} = kaboose:timesup(Aroom),
Points = maps:get("Konrad", Dict),
?assertMatch(500 ,Points).
```

Verification of timer working was performed by checking the number of points that were awarded to the user providing a right answer.

Correctness of Solution

Based on the tests, analyzing the code we assume the solution is likely to be correct.
Additionally, the solutions passed the Online TA test.

Summary of Code Quality

Based on the tests, we do believe that the functionality of the kaboose server matches the functionality specified by the assignment.

Additionally, we think that our code is very well structured by using three different modules to implement the functionality and extracting the non functional code to the `basic_server` module. This leads a better maintainability of the code.

One of the major downsides of this solution is the "perfect world" assumption. By not checking the format of input data, it is very easy to produce a runtime error by inputting ill-formatted data. As we do not implement a Supervisor/Worker pattern in this solution, the application can not recover from this state.

Appendices

A Assignment 1

A.1 actionModuleServer.erl

```erlang
%%%-------------------------------------------------------------------
%%% This module turns an actionModule into a server
%%% based on the gen_server Example by Ken Friis Larsen (see below)
%%% @author Per Steffen Czolbe
%%%-------------------------------------------------------------------




%%%-------------------------------------------------------------------
%%% @author Ken Friis Larsen <ken@friislarsen.net>
%%% @copyright (C) 2017, Ken Friis Larsen
%%% @doc
%%%
%%% @end
%%% Created : 12 Oct 2017 by Ken Friis Larsen <ken@friislarsen.net>
%%%-------------------------------------------------------------------
-module(actionModuleServer).

%% needs action module with these callbacks:
-callback initialise(Arg :: term()) ->
    { ok, State :: term()} |
    { error, Reason :: term()}.
-callback action(Req :: term(), Env :: term(), State :: term()) ->
    {new_state, Content :: term(), NewState :: term()} |
    {no_change, Content :: term()}.

%% uses gen_server
-behaviour(gen_server).
%% uses simpleSupervisor
-behaviour(simpleSupervisor).
```

```
%% API
-export([start/2, action/3, start_link/1]).


%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

% define ?SERVER = ?MODULE = modulename
-define(SERVER, ?MODULE).


%%%===================================================================
%%% API
%%%===================================================================

% ServerRef can be the Pid
action(ServerRef, Request, Enviroment) ->
    gen_server:cast(ServerRef, {self(), {action, Request, Enviroment}}),
    receive
        {_From, worker_died} -> {error, 500};
        {_From, Content} -> Content
    end.

%%--------------------------------------------------------------------
%% @doc
%% Starts the server
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
%% @end
%%--------------------------------------------------------------------
start(ActionModule, Args) ->
    simpleSupervisor:start(?MODULE, [ActionModule, Args]).


start_link([ActionModule, Args]) ->
    gen_server:start_link(?MODULE, [ActionModule, Args], []).  % Module,
 ↪  args, options

%%%===================================================================
%%% gen_server callbacks
%%%===================================================================

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Initializes the server
%%
%% @spec init(Args) -> {ok, State} |
%%                     {ok, State, Timeout} |
%%                      ignore |
%%                      {stop, Reason}
%% @end
%%--------------------------------------------------------------------
init([ActionModule, Args]) ->
    case ActionModule:initialise(Args) of
        {ok, State} -> {ok, {ActionModule, State}};
        {error, Reason} -> {stop, Reason}
    end.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling call messages
%%
%% @spec handle_call(Request, From, State) ->
%%                                   {reply, Reply, State} |
%%                                   {reply, Reply, State, Timeout} |
%%                                   {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, Reply, State} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
    {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling cast messages
%%
%% @spec handle_cast(Msg, State) -> {noreply, State} |
%%                                  {noreply, State, Timeout} |
%%                                  {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_cast({From, {action, Request, Enviroment}}, State) ->
    {ActionModule, ModuleState} = State,
    case ActionModule:action(Request, Enviroment, ModuleState) of
        {new_state, Content, NewState} ->
            reply(From, Content),
            {noreply, {ActionModule, NewState}};
        {no_change, Content} ->
            reply(From, Content),
            {noreply, State}
    end.

reply(To, Response) ->
    To ! {self(), Response}.
```

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling all non call/cast messages
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
    {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any
%% necessary cleaning up. When it returns, the gen_server terminates
%% with Reason. The return value is ignored.
%%
%% @spec terminate(Reason, State) -> void()
%% @end
%%--------------------------------------------------------------------
terminate(_Reason, _State) ->
    ok.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Convert process state when code is changed
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

%%%===================================================================
%%% Internal functions
%%%===================================================================
```

A.2 simpleSupervisor.erl

```
%% Simple supervisor. Does not keep track of child state, child is
 ↪  restarted in initial state.
%% inc ase of worker crash, send message {SupervisorPid, worker_died} to
 ↪  last message sender
```

```erlang
-module(simpleSupervisor).
-export([start/2]).

%% needs a worker module with these callbacks:
-callback start_link(Args :: term()) ->
    { ok, State :: term()} |
    { error, Reason :: term()}.


%% Interface

% starts the supervisor, which in turn starts the given module with the
 ↪  args
% returns Pid used to communicate with the worker
start(WorkerModule, WorkerArgs) ->
    SPid = spawn(fun () ->
                    process_flag(trap_exit, true),
                    {ok, WorkerPid} = spawnWorker(WorkerModule,
 ↪  WorkerArgs),
                    supervisor(WorkerPid, nothing, WorkerModule,
 ↪  WorkerArgs)
                end),
    case SPid of
        [] -> {error, failed_to_spawn_supervisor};
        _ -> {ok, SPid}
    end.


%% Internal implementation

spawnWorker(Module, Args) ->
    Module:start_link(Args).


supervisor(SlavePid, LastMsg, RestartModule, RestartArgs) ->
    receive
        % worker died
        {’EXIT’, SlavePid, Reason} ->
            io:format("~p exited because of ~p~n", [SlavePid, Reason]),
            {ok, Pid} = spawnWorker(RestartModule, RestartArgs),
            case LastMsg of
                {_, {From, _}} ->
                    From ! {self(), worker_died};
                Message -> io:format("Supervisor coudnt decipher msg
 ↪  ~p~n", [Message])
            end,
            supervisor(Pid, nothing, RestartModule, RestartArgs);
        % someone wants to msg worker
        Msg ->
            SlavePid ! Msg,
            supervisor(SlavePid, Msg, RestartModule, RestartArgs)
    end.
```