Advanced Programming Assignment 6: OTP Report

AP22 Assignment 6 Group 60
KRW521, CPJ395

October 25, 2022

1 Design and Implementation

After reading the assignment, we find out that it is a good idea to split the code into two parts, one for broker and the other for coordinator, because these two are two characters in this assignment, and they have their own features. Broker is like the server that handle the request from the players (clients), so we write **rps.erl** for **broker** with **gen_server** as the **behaviour** in it. While a coordinator is like a judge for a game, so we write **rps_coor.erl** for **coordinator** with **gen_statem** as the **behaviour** in it. For a quite simple example (as shown in the figure below, dotted line is just for identifying different **erl** files), there are 4 processes running, two of which are for players, one for broker (gen_server) and one for coordinator(gen_statem). And they communicate in the ways as shown in **Figure 1**.

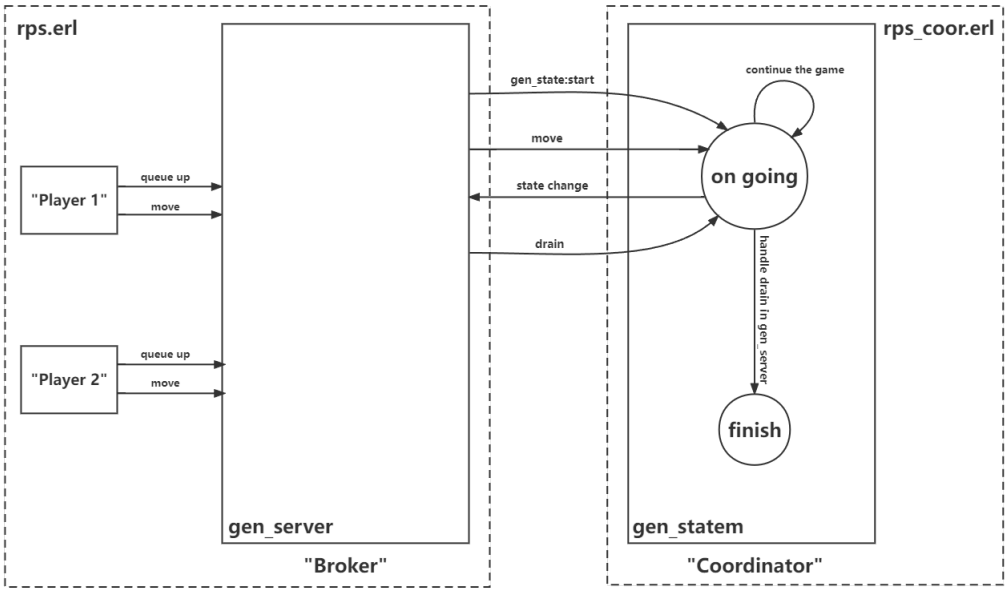

Figure 1: A "Broker" and a "coordinator" for 2 "Players"

For the broker it has **handle_call** and **handle_cast** functions to handle the request from players (like queuing and move) and from coordinators (like after finishing a game the coordinator will ask the broker to record some information about this game in broker's **State**). For the coordinator, it has 2 state_functions **ongoing** and **finished**, and what they receive and how they transform are shown in **Figure 1**.
When the number of players increases, there may be more coordinator to serve them, and the relation graph is shown in **Figure 2** (detail information

is omitted).

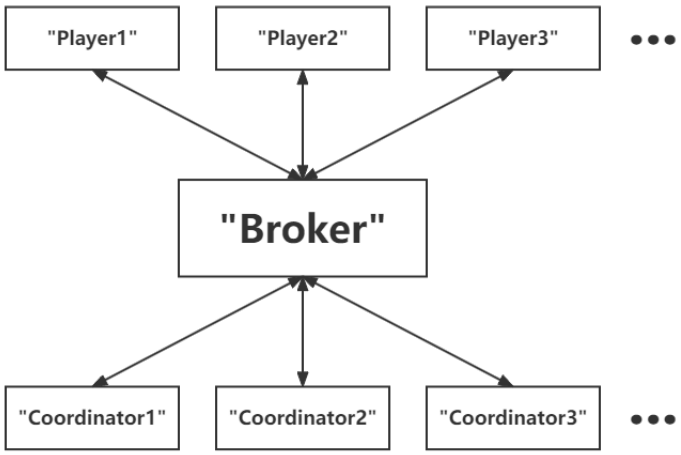

Figure 2: A "Broker" and many "coordinators" for many "Players"

As for the states **broker** and **coordinator** maintain, for broker, it maintain a tuple: {**StateFlag, Queue, FinishedCoorsRounds, OngoingCoors**} where **StateFlag** denotes the broker is active or drain, **Queue** is the list that contaiins players waiting in queue (for each player we use a tuple to record him(or her): {**Name, Rounds, Pid of the player**}), **FinishedCoorsRounds** is a list that contains the records of total rounds of every finished game, and **OngoingCoors** is also a list to contain the Pids of ongoing coordinators, and for coordinator, it also maintain a tuple: {**Participants, WaitP, ChoicesThisRound, RoundsWin1, RoundsWin2, TotalRounds, MaxRounds, BrokerRef** }, where **Participants** is a list to contain Pids of two players in a game, **WaitP** can be an atom **nobody** or the Pid of a player denotes that the State of coordinator is visited by a player or not(for example, it can be to the Pid of the player who gives the **Choice** first, when the second player gives his/her **Choice**, the program can send the reply to the blocking **call** of the player who gives the **Choice** first), **ChoicesThisRound** is the list contains the Choices given by the two players, **RoundsWin1** denotes the winning rounds of the first player in **Participants** list, **RoundsWin2** denotes the winning rounds of the second player in **Participants** list, **TotalRounds** denotes the total rounds of this game, **MaxRounds** denotes the maximum number of rounds of the game, and **BrokerRef** denotes the Pid of the broker who assigns the coordinators.
Also, this time we have two helper functions: **judge_winner** to judge the **Choices** in a list and return the index of the Choice that wins this round,

and **find_player_order** to find the index of the player in a list.

2 Assessment of The Code

2.1 Completeness

All functions are completed, and the completion of all functions are as follows:

Class of Function	Function Name	Completion
gen_server api	start	Completed
gen_server api	queue_up	Completed
gen_server api	move	Completed
gen_server api	statistics	Completed
gen_server api	drain	Completed
gen_server api	get_server_state	Completed
gen_server api	delete_coor	Completed
gen_server api	add_total_rounds	Completed
gen_server callback function	init	Completed
gen_server callback function	handle_call	Completed
gen_server callback function	handle_cast	Completed
gen_statem api	start	Completed
gen_statem api	coor_move	Completed
gen_statem api	coor_finish	Completed
gen_statem api	coor_register_player	Completed
gen_statem state_function	ongoing	Completed
gen_statem state_function	finished	Completed
gen_statem callback function	init	Completed
gen_statem callback function	callback_mode	Completed
helper functions	judge_winner	Completed
helper functions	find_player_order	Completed

2.2 Correctness

This time our code performs differently in online TA and in our test set, and the online TA gives a quite weird result while executing our test set, and that result is different from the testing result while we test on our computers. In fact, our code perform well on our testing set(as shown in figure 3, and you can have a look at out testing set. What's more we use a **rock bot** and a **paper bot** in our test set(we write the **paper bot** in the same way as the

rock bot)). So we can have the correctness table:

Class of Function	Function Name	Test Result
gen_server api	start	OK
gen_server api	queue_up	OK
gen_server api	move	not bad
gen_server api	statistics	not bad
gen_server api	drain	OK
gen_server api	get_server_state	OK
gen_server api	delete_coor	OK
gen_server api	add_total_rounds	OK
gen_server callback function	init	OK
gen_server callback function	handle_call	OK
gen_server callback function	handle_cast	OK
gen_statem api	start	OK
gen_statem api	coor_move	OK
gen_statem api	coor_finish	OK
gen_statem api	coor_register_player	OK
gen_statem state_function	ongoing	OK
gen_statem state_function	finished	OK
gen_statem callback function	init	OK
gen_statem callback function	callback_mode	OK
helper functions	judge_winner	OK
helper functions	find_player_order	OK

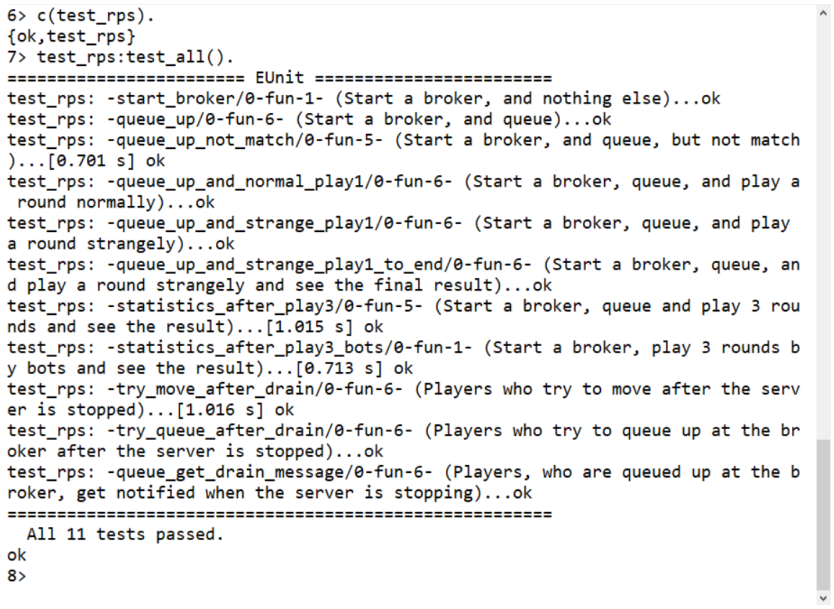

Figure 3: Output of our code testing on our computers

2.3 Efficiency

The efficiency of our program this time is not bad, every function can response in a relatively short time.

2.4 Robustness

The Robustness of our program is also at a high level.

2.5 Maintainability

This time, because we divide the code into two parts(two **erl** files): **gens_server** for **Broker** and **gen_statem** for **Coordinator**, we believe that our code can have good maintainability.

Class of Function	Function Name	Maintainability
gen_server api	start	Good
gen_server api	queue_up	Good
gen_server api	move	Good
gen_server api	statistics	Good
gen_server api	drain	Good
gen_server api	get_server_state	Good
gen_server api	delete_coor	Good
gen_server api	add_total_rounds	Good
gen_server callback function	init	Good
gen_server callback function	handle_call	Good
gen_server callback function	handle_cast	Good
gen_statem api	start	Good
gen_statem api	coor_move	Good
gen_statem api	coor_finish	Good
gen_statem api	coor_register_player	Good
gen_statem state_function	ongoing	Good
gen_statem state_function	finished	Good
gen_statem callback function	init	Good
gen_statem callback function	callback_mode	Good
helper functions	judge_winner	Good
helper functions	find_player_order	Good

A Appendix: rps.erl

```
1  -module(rps).
2  -export([start/0, queue_up/3, move/2, statistics/1, drain/3,
   ↪  get_server_state/1, delete_coor/2, add_total_rounds/2]).
3
4
5
6
7  -export([init/1, handle_call/3, handle_cast/2]).
8  -behaviour(gen_server).
9
10
11 start() -> gen_server:start(?MODULE,{active, [], [0], []},[]).
12
13 queue_up(BrokerRef, Name, Rounds) ->
14     ServerState = get_server_state(BrokerRef),
15     Me = self(),
16     if
17         ServerState =:= drain -> error_server_stopping;
18         Rounds < 0 -> {error, rounds_should_be_non_negative};
19         Rounds >= 0 ->
20             {OtherPlayer, Coordinator} =
   ↪  gen_server:call(BrokerRef,{queue_up_trans, Name, Rounds}),
21             rps_coor:coor_register_player(BrokerRef, Coordinator, Me),
22             {ok, OtherPlayer, Coordinator}
23     end.
24
25 move(Coordinator, Choice) ->
26     rps_coor:coor_move(Coordinator, Choice).
27
28 statistics(BrokerRef) -> gen_server:call(BrokerRef, statistics_trans).
29
30 drain(BrokerRef, Pid, Msg) ->
31     if
32         Pid =/= none ->
33             Pid ! Msg,
34             gen_server:cast(BrokerRef, drain_trans);
35         Pid =:= none ->
36             gen_server:cast(BrokerRef, drain_trans)
```

```
37      end.
38
39  init({StateFlag, Queue, FinishedCoorsRounds, OngoingCoors}) ->
40      {ok, {StateFlag, Queue, FinishedCoorsRounds, OngoingCoors}}.
41
42  handle_call({queue_up_trans, Name, Rounds}, _From, State) ->
43      case State of
44          {active, Queue, FinishedCoorsRounds, OngoingCoors} ->
45              FindMatchResult = lists:keyfind(Rounds, 2, Queue),
46              case FindMatchResult of
47                  false ->
48                      NewQueue = [{Name, Rounds, _From}]++Queue,
49                      {noreply, {active, NewQueue, FinishedCoorsRounds,
    ↪  OngoingCoors}};
50                  {OtherPlayer, _, _OpFrom} ->
51                      {ok, Coordinator} = rps_coor:start(Rounds), % Rounds
    ↪  is also MaxRounds in "rps_coor.erl"
52                      gen_server:reply(_OpFrom, {Name, Coordinator}),
53                      NewQueue = lists:delete(FindMatchResult, Queue),
54                      NewOngoingCoors = [Coordinator|OngoingCoors],
55                      {reply, {OtherPlayer, Coordinator}, {active,
    ↪  NewQueue, FinishedCoorsRounds, NewOngoingCoors}}
56              end
57      end;
58  handle_call(statistics_trans, _From, State) ->
59      case State of
60          {active, Queue, FinishedCoorsRounds, OngoingCoors} ->
61              LongestGame = lists:max(FinishedCoorsRounds),
62              InQueueNum = length(Queue),
63              OngoingCoorsNum = length(OngoingCoors),
64              {reply, {ok, LongestGame, InQueueNum, OngoingCoorsNum},
    ↪  State}
65      end;
66  handle_call(get_server_state_trans, _From, State) ->
67      case State of
68          {active, _, _, _} -> {reply, active, State};
69          {drain, _, _, _} -> {reply, drain, State}
70      end.
71
72
```

```
73
74  handle_cast(drain_trans, State) ->
75      case State of
76          {active, Queue, FinishedCoorsRounds, OngoingCoors} ->
77              Fun = fun(Coordinator) -> rps_coor:coor_finish(Coordinator)
    ↪  end,
78              lists:foreach(Fun, OngoingCoors),
79              FunStop = fun (ElemTuple)->
80                  case ElemTuple of
81                      {_, _, Pid} -> gen_server:reply(Pid,
    ↪  error_server_stopping)
82                  end end,
83              lists:foreach(FunStop, Queue),
84              {noreply, {drain, Queue, FinishedCoorsRounds, OngoingCoors}};
85          _ -> {noreply, State}
86      end;
87  handle_cast({delete_coor_trans, Coordinator},State) ->
88      case State of
89          {StateFlag, Queue, FinishedCoorsRounds, OngoingCoors} ->
90              NewOngoingCoors = lists:delete(Coordinator, OngoingCoors),
91              {noreply, {StateFlag, Queue, FinishedCoorsRounds,
    ↪  NewOngoingCoors}}
92      end;
93  handle_cast({add_total_rounds_trans, TotalRounds},State) ->
94      case State of
95          {StateFlag, Queue, FinishedCoorsRounds, OngoingCoors} ->
96              NewFinishedCoorsRounds = [TotalRounds|FinishedCoorsRounds],
97              {noreply, {StateFlag, Queue, NewFinishedCoorsRounds,
    ↪  OngoingCoors}}
98      end.
99
100
101 %some api functions
102
103 get_server_state(BrokerRef) ->
104     gen_server:call(BrokerRef, get_server_state_trans).
105
106 delete_coor(BrokerRef, Coordinator) ->
107     gen_server:cast(BrokerRef, {delete_coor_trans, Coordinator}).
108
```

```
109 add_total_rounds(BrokerRef, TotalRounds) ->
110     gen_server:cast(BrokerRef, {add_total_rounds_trans, TotalRounds}).
111
```

B Appendix: rps_coor.erl

```
-module(rps_coor).

% api called by rps.erl
-export([start/1, coor_move/2, coor_finish/1, coor_register_player/3]).



-export([callback_mode/0, init/1, ongoing/3, finished/3]).
-behaviour(gen_statem).



start(MaxRounds) ->
    gen_statem:start(?MODULE,{[], nobody, [], 0, 0, 0, MaxRounds,
    ↪  none},[]).


coor_move(Coordinator, Choice) ->
    gen_statem:call(Coordinator, {coor_move_trans, Choice, Coordinator}).
%gen_statem:call will wait for a process to use reply(From, Reply) in
    ↪  StateName_function
%Only after reply(From, Reply) in StateName_function is used by a
    ↪  process, can the program here continue to execute.

coor_finish(Coordinator) -> % drain will call this
    gen_statem:cast(Coordinator, coor_finish_trans).

coor_register_player(BrokerRef, Coordinator, PPid)->
    gen_statem:cast(Coordinator, {coor_register_player_trans, BrokerRef,
    ↪  PPid}).




callback_mode() -> state_functions.

init({Participants, WaitP, ChoicesThisRound, RoundsWin1, RoundsWin2,
    ↪  TotalRounds, MaxRounds, BrokerRef}) ->
```

```
34      {ok, ongoing, {Participants, WaitP, ChoicesThisRound, RoundsWin1,
    ↪  RoundsWin2, TotalRounds, MaxRounds, BrokerRef}}.
35
36  ongoing(cast,{coor_register_player_trans, BrokerRef, PPid}, State) ->
37      case State of
38          {Participants, WaitP, ChoicesThisRound, RoundsWin1, RoundsWin2,
    ↪  TotalRounds, MaxRounds, _} ->
39              NewParticipants = [PPid|Participants],
40              {keep_state, {NewParticipants, WaitP, ChoicesThisRound,
    ↪  RoundsWin1, RoundsWin2, TotalRounds, MaxRounds, BrokerRef}}
41      end;
42  ongoing({call, From}, {coor_move_trans, Choice, Coordinator},
    ↪  {Participants, WaitP, ChoicesThisRound, RoundsWin1, RoundsWin2,
    ↪  TotalRounds, MaxRounds, BrokerRef})->
43      if
44          ((TotalRounds == MaxRounds) or (RoundsWin1 > (MaxRounds/2)) or
    ↪  (RoundsWin2 > (MaxRounds/2))) ->
45              {RealPid,_}=From,
46              YourOrder = find_player_order(RealPid, Participants),
47              case YourOrder of
48                  1 ->
49                      case WaitP of
50                          nobody ->
51                              gen_statem:reply(From, {game_over,
    ↪  RoundsWin1, RoundsWin2}),
52                              {keep_state, {Participants, From,
    ↪  ChoicesThisRound, RoundsWin1, RoundsWin2, TotalRounds, MaxRounds,
    ↪  BrokerRef}};
53                          _ ->
54                              gen_statem:reply(From, {game_over,
    ↪  RoundsWin1, RoundsWin2}),
55                              rps:delete_coor(BrokerRef, Coordinator),
56                              rps:add_total_rounds(BrokerRef, TotalRounds),
57                              {next_state, finished, {Participants, WaitP,
    ↪  ChoicesThisRound, RoundsWin1, RoundsWin2, TotalRounds, MaxRounds,
    ↪  BrokerRef}}
58                      end;
59                  2 ->
60                      case WaitP of
61                          nobody ->
```

```
62                              gen_statem:reply(From, {game_over,
↪ RoundsWin2, RoundsWin1}),
63                              {keep_state, {Participants, From,
↪ ChoicesThisRound, RoundsWin1, RoundsWin2, TotalRounds, MaxRounds,
↪ BrokerRef}};
64                          _ ->
65                              gen_statem:reply(From, {game_over,
↪ RoundsWin2, RoundsWin1}),
66                              rps:delete_coor(BrokerRef, Coordinator),
67                              rps:add_total_rounds(BrokerRef, TotalRounds),
68                              {next_state, finished, {Participants, WaitP,
↪ ChoicesThisRound, RoundsWin1, RoundsWin2, TotalRounds, MaxRounds,
↪ BrokerRef}}
69                      end
70              end;
71          TotalRounds < MaxRounds ->
72              case WaitP of
73                  nobody -> %you are the first one to give the choice
74                      NewChoicesThisRound = [Choice|ChoicesThisRound],
75                      {keep_state,  {Participants, From,
↪ NewChoicesThisRound, RoundsWin1, RoundsWin2, TotalRounds, MaxRounds,
↪ BrokerRef}};
76                  _ -> %you are the second one to give the choice, so you
↪ should update the result
77                      {RealPid,_}=From,
78                      YourOrder = find_player_order(RealPid, Participants),
↪ %get player order
79                      case YourOrder of
80                          1 ->
81                              NewChoicesThisRound =
↪ [Choice|ChoicesThisRound],
82                              Winner = judge_winner(NewChoicesThisRound),
83                              case Winner of
84                                  1 ->
85                                      gen_statem:reply(From, win),
86                                      gen_statem:reply(WaitP, {loss,
↪ Choice}),
87                                      {keep_state,{Participants, nobody,
↪ [], RoundsWin1+1, RoundsWin2, TotalRounds+1, MaxRounds, BrokerRef}};
88                                  2 ->
```

```
89                                      OpChoice =
    ↪ lists:nth(2,NewChoicesThisRound),
90                                      gen_statem:reply(From, {loss,
    ↪ OpChoice}),
91                                      gen_statem:reply(WaitP, win),
92                                      {keep_state,{Participants, nobody,
    ↪ [], RoundsWin1, RoundsWin2+1, TotalRounds+1, MaxRounds, BrokerRef}};
93                                  3 ->
94                                      gen_statem:reply(From, tie),
95                                      gen_statem:reply(WaitP, tie),
96                                      {keep_state,{Participants, nobody,
    ↪ [], RoundsWin1, RoundsWin2, TotalRounds+1, MaxRounds, BrokerRef}}
97                              end;
98                          2 ->
99                              NewChoicesThisRound =
    ↪ [Choice|ChoicesThisRound],
100                             Winner = judge_winner(NewChoicesThisRound),
101                             case Winner of
102                                 1 ->
103                                     gen_statem:reply(From, win),
104                                     gen_statem:reply(WaitP, {loss,
    ↪ Choice}),
105                                     {keep_state,{Participants, nobody,
    ↪ [], RoundsWin1, RoundsWin2+1, TotalRounds+1, MaxRounds, BrokerRef}};
106                                 2 ->
107                                     OpChoice =
    ↪ lists:nth(2,NewChoicesThisRound),
108                                     gen_statem:reply(From, {loss,
    ↪ OpChoice}),
109                                     gen_statem:reply(WaitP, win),
110                                     {keep_state,{Participants, nobody,
    ↪ [], RoundsWin1+1, RoundsWin2, TotalRounds+1, MaxRounds, BrokerRef}};
111                                 3 ->
112                                     gen_statem:reply(From, tie),
113                                     gen_statem:reply(WaitP, tie),
114                                     {keep_state,{Participants, nobody,
    ↪ [], RoundsWin1, RoundsWin2, TotalRounds+1, MaxRounds, BrokerRef}}
115                             end
116                     end
117             end
```

```
118     end;
119 ongoing(cast, coor_finish_trans, State) ->
120     {next_state, finished, State}.
121 
122 finished({call, From},{coor_move_trans, _, _}, State) ->
123     gen_statem:reply(From, server_stopping),
124     {next_state, finished, State}.
125 
126 %some helper functions
127 judge_winner(GestureList) ->
128     case GestureList of
129         [rock,paper] -> 2;
130         [rock,scissors] -> 1;
131         [rock,rock] -> 3;
132         [paper,rock] -> 1;
133         [paper,paper] -> 3;
134         [paper,scissors] -> 2;
135         [scissors,rock] -> 2;
136         [scissors,paper] -> 1;
137         [scissors,scissors] -> 3;
138         [rock,_] -> 1;
139         [paper,_] -> 1;
140         [scissors,_] -> 1;
141         [_,rock] -> 2;
142         [_,paper] -> 2;
143         [_,scissors] -> 2;
144         [_,_] -> 3
145     end.
146 
147 find_player_order(PPid, Participants) ->
148     Player1 = lists:nth(1, Participants),
149     Player2 = lists:nth(2, Participants),
150     if
151         PPid == Player1 -> 1;
152         PPid == Player2 -> 2
153     end.
```

C Appendix: paper_bot.erl

```
-module(paper_bot).
-export([queue_up_and_play/1]).

queue_up_and_play(Broker) ->
    {ok, _Other, Coor} = rps:queue_up(Broker, "Paper bot(amy)", 3),
    paper_to_game_over(Coor).

paper_to_game_over(Coor) ->
    case rps:move(Coor, paper) of
        {game_over, Me, SomeLoser} ->
            {ok, Me, SomeLoser};
        server_stopping ->
            server_stopping;
        _ -> paper_to_game_over(Coor)
    end.

```

D Appendix: test_rps.erl

```
-module(test_rps).
-export([test_all/0]).

%% Maybe you want to use eunit
-include_lib("eunit/include/eunit.hrl").


test_all() ->
    eunit:test(
      [
       start_broker(),
       queue_up(),
       queue_up_not_match(),
       queue_up_and_normal_play1(),
       queue_up_and_strange_play1(),
       queue_up_and_strange_play1_to_end(),
       statistics_after_play3(),
       statistics_after_play3_bots(),
       try_move_after_drain(),
       try_queue_after_drain(),
       queue_get_drain_message()
      ], [verbose]).

start_broker() ->
  {"Start a broker, and nothing else",
    fun() ->
      % one kind of assertMatch(at the end of "fun()"")
      ?assertMatch({ok, _}, rps:start())
    end}.

queue_up() ->
  {"Start a broker, and queue",
    fun() ->
      {ok, S}=rps:start(),
      Pid = self(),
      TestProcess1 = fun(PidT, ST) ->
        Res = rps:queue_up(ST,"Bob",1),
        % Me=self(),
```

```
39          PidT ! {p1,Res}  end,
40        TestProcess2 = fun(PidT, ST) ->
41          Res = rps:queue_up(ST,"Amy",1),
42          %Me=self(),
43          PidT ! {p2,Res}  end,
44        spawn(fun() -> TestProcess1(Pid, S) end),
45        spawn(fun() -> TestProcess2(Pid, S) end),
46        receive
47          {p1, Res} ->
48            % ?assertMatch (?assertEqual) can also be written in this way
49            ?assertMatch({ok, "Amy", _}, Res),
50            receive
51              {p2, Res1} ->
52                ?assertMatch({ok, "Bob", _}, Res1)
53            end
54        end
55      end}.
56
57  queue_up_not_match() ->
58    {"Start a broker, and queue, but not match",
59      fun() ->
60        {ok, S}=rps:start(),
61        TestProcess1 = fun(ST) ->
62          rps:queue_up(ST,"Bob",3)
63          end,
64        TestProcess2 = fun(ST) ->
65          rps:queue_up(ST,"Amy",1)
66          end,
67        spawn(fun() -> TestProcess1(S) end),
68        spawn(fun() -> TestProcess2(S) end),
69        timer:sleep(700),
70        Res=rps:statistics(S),
71        ?assertEqual({ok, 0, 2, 0}, Res)
72      end}.
73
74  queue_up_and_normal_play1() ->
75    {"Start a broker, queue, and play a round normally",
76      fun() ->
77        {ok, S}=rps:start(),
78        Pid = self(),
```

```
79        TestProcess1 = fun(PidT, ST) ->
80          {ok, _, Coordinator} = rps:queue_up(ST,"Bob",1),
81          Res=rps:move(Coordinator, rock),
82          %Me=self(),
83          PidT ! {p1,Res}  end,
84        TestProcess2 = fun(PidT, ST) ->
85          {ok, _, Coordinator} = rps:queue_up(ST,"Amy",1),
86          Res=rps:move(Coordinator, paper),
87          %Me=self(),
88          PidT ! {p2,Res}  end,
89        spawn(fun() -> TestProcess1(Pid, S) end),
90        spawn(fun() -> TestProcess2(Pid, S) end),
91        receive
92          {p1, Res} ->
93            ?assertEqual({loss, paper}, Res),
94            receive
95              {p2, Res1} ->
96                ?assertEqual(win, Res1)
97            end
98        end
99      end}.
100
101 queue_up_and_strange_play1() ->
102   {"Start a broker, queue, and play a round strangely",
103     fun() ->
104       {ok, S}=rps:start(),
105       Pid = self(),
106       TestProcess1 = fun(PidT, ST) ->
107         {ok, _, Coordinator} = rps:queue_up(ST,"Bob",1),
108         Res=rps:move(Coordinator, laser),
109         %Me=self(),
110         PidT ! {p1,Res}  end,
111       TestProcess2 = fun(PidT, ST) ->
112         {ok, _, Coordinator} = rps:queue_up(ST,"Amy",1),
113         Res=rps:move(Coordinator, paper),
114         %Me=self(),
115         PidT ! {p2,Res}  end,
116       spawn(fun() -> TestProcess1(Pid, S) end),
117       spawn(fun() -> TestProcess2(Pid, S) end),
118       receive
```

```
119         {p1, Res} ->
120           ?assertEqual({loss, paper}, Res),
121           receive
122             {p2, Res1} ->
123               ?assertEqual(win, Res1)
124           end
125       end
126     end}.
127
128 queue_up_and_strange_play1_to_end() ->
129   {"Start a broker, queue, and play a round strangely and see the final
      ↪  result",
130     fun() ->
131       {ok, S}=rps:start(),
132       Pid = self(),
133       TestProcess1 = fun(PidT, ST) ->
134         {ok, _, Coordinator} = rps:queue_up(ST,"Bob",1),
135         rps:move(Coordinator, laser),
136         Res= rps:move(Coordinator, thumbs_up),
137         %Me=self(),
138         PidT ! {p1,Res}  end,
139       TestProcess2 = fun(PidT, ST) ->
140         {ok, _, Coordinator} = rps:queue_up(ST,"Amy",1),
141         rps:move(Coordinator, paper),
142         Res=rps:move(Coordinator, rock),
143         %Me=self(),
144         PidT ! {p2,Res}  end,
145       spawn(fun() -> TestProcess1(Pid, S) end),
146       spawn(fun() -> TestProcess2(Pid, S) end),
147       receive
148         {p1, Res} ->
149           ?assertEqual({game_over, 0, 1}, Res),
150           receive
151             {p2, Res1} ->
152               ?assertEqual({game_over, 1, 0}, Res1)
153           end
154       end
155     end}.
156
157 statistics_after_play3() ->
```

```
158   {"Start a broker, queue and play 3 rounds and see the result",
159     fun() ->
160       {ok, S}=rps:start(),
161       TestProcess1 = fun(ST) ->
162         {ok, _, Coordinator} = rps:queue_up(ST,"Bob",3),
163         rps:move(Coordinator, laser),
164         rps:move(Coordinator, rock),
165         rps:move(Coordinator, scissors)
166         end,
167       TestProcess2 = fun(ST) ->
168         {ok, _, Coordinator} = rps:queue_up(ST,"Amy",3),
169         rps:move(Coordinator, paper),
170         rps:move(Coordinator, paper),
171         rps:move(Coordinator, scissors)
172         end,
173       spawn(fun() -> TestProcess1(S) end),
174       spawn(fun() -> TestProcess2(S) end),
175       timer:sleep(1000),
176       Res=rps:statistics(S),
177       ?assertEqual({ok, 2, 0, 0},Res)
178     end}.
179
180 statistics_after_play3_bots() ->
181   {"Start a broker, play 3 rounds by bots and see the result",
182     fun() ->
183       {ok, S}=rps:start(),
184       spawn(rock_bot, queue_up_and_play, [S]),
185       spawn(paper_bot, queue_up_and_play, [S]),
186       timer:sleep(700),
187       Res=rps:statistics(S),
188       ?assertEqual({ok, 2, 0, 0},Res)
189     end}.
190
191 try_move_after_drain() ->
192   {"Players who try to move after the server is stopped",
193     fun() ->
194       {ok, S}=rps:start(),
195       Pid = self(),
196       TestProcess1 = fun(PidT,ST) ->
197         {ok, _, Coordinator}=rps:queue_up(ST,"Bob",1),
```

```
198         timer:sleep(1000),
199         Res=rps:move(Coordinator, rock),
200         PidT ! {p1, Res}
201         end,
202       TestProcess2 = fun(PidT,ST) ->
203         {ok, _, Coordinator}=rps:queue_up(ST,"Amy",1),
204         timer:sleep(1000),
205         Res=rps:move(Coordinator, rock),
206         PidT ! {p2, Res}
207         end,
208       spawn(fun() -> TestProcess1(Pid,S) end),
209       spawn(fun() -> TestProcess2(Pid,S) end),
210       timer:sleep(500),
211       rps:drain(S,none,none),
212       receive
213         {p1, Res} ->
214           ?assertEqual(server_stopping, Res),
215           receive
216             {p2, Res1} ->
217               ?assertEqual(server_stopping, Res1)
218           end
219       end
220     end}.
221
222 try_queue_after_drain() ->
223   {"Players who try to queue up at the broker after the server is
      ↪  stopped",
224     fun() ->
225       {ok, S}=rps:start(),
226       Pid = self(),
227       TestProcess1 = fun(PidT,ST) ->
228         Res=rps:queue_up(ST,"Bob",3),
229         PidT ! {p1, Res}
230         end,
231       TestProcess2 = fun(PidT,ST) ->
232         Res=rps:queue_up(ST,"Amy",1),
233         PidT ! {p2, Res}
234         end,
235       rps:drain(S,none,none),
236       spawn(fun() -> TestProcess1(Pid,S) end),
```

```
237       spawn(fun() -> TestProcess2(Pid,S) end),
238       receive
239         {p1, Res} ->
240           ?assertEqual(error_server_stopping, Res),
241           receive
242             {p2, Res1} ->
243               ?assertEqual(error_server_stopping, Res1)
244           end
245       end
246     end}.
247
248 queue_get_drain_message() ->
249   {"Players, who are queued up at the broker, get notified when the
      ↪  server is stopping",
250     fun() ->
251       {ok, S}=rps:start(),
252       Pid = self(),
253       TestProcess1 = fun(PidT,ST) ->
254         Res=rps:queue_up(ST,"Bob",3),
255         PidT ! {p1, Res}
256         end,
257       TestProcess2 = fun(PidT,ST) ->
258         Res=rps:queue_up(ST,"Amy",1),
259         PidT ! {p2, Res}
260         end,
261       spawn(fun() -> TestProcess1(Pid,S) end),
262       spawn(fun() -> TestProcess2(Pid,S) end),
263       rps:drain(S,none,none),
264       receive
265         {p1, Res} ->
266           ?assertEqual(error_server_stopping, Res),
267           receive
268             {p2, Res1} ->
269               ?assertEqual(error_server_stopping, Res1)
270           end
271       end
272     end}.
273
```