Quizmaster

Philip Rajani Lassen (vgh804)
Nicolas Ringsmose Larsen (vgn209)

October 2018

1 Implementation

We began by implementing the start function. The start function calls `spawn` for a new quizmaster process and invokes the `loop` function. The `loop` function handles the majority of the logic for the server. This is the same method used in Flamingo as well, and is the method that was discussed in lecture for implementing a server.

In order to implement the remaining functions we call `loop` with three arguments. The first is the variable `Questions` which stores the questions and is talked about in the next paragraph. Then we store a map of `Players` that have joined the game by a reference id that is given to them as they join. This is important as we need to broadcast results and new questions to the players that have joined the game. Then finally, we also store status. Status allows us to encode different stages and states of the game so that when we call certain functions, we can throw errors or process the requests appropriately.

Next we implemented the `add_question` and the `get_questions` functions. We use a Queue to store the questions. This way we can keep the ordering of the questions (something that the specification requires). After testing that the `add_question` and the `get_questions` worked properly, we moved onto the `play` function.

`play` starts a new quiz if there is a question. It also checks to make sure that the game hasn't already been started by another conductor to ensure that only one conductor controls the whole game. We allow the first person to call `play` to become the conductor.

The `next` function begins the next question and returns a tuple with the description of the question as well as a list of possible answers, with the correct one hidden. The question is also broadcast to all players that have joined the game. Status is something that our server keeps track of in order to make sure that there is no active question when `next` is called. Otherwise we make sure to throw an error. Again all this logic is encoded in the loop function.

In the `loop`, the `join` function is handled by checking that the Status of the quiz still allows players to be joined. Then if the name of the player hasn't been taken, the player is added to the map of players. We have decided to only let players join the quiz if the status of the quiz is `editable`, as any other status seems to indicate that the game has already started, so that made the most sense to us.

The `leave` function is very simple. When the `loop` function receives a message that pattern matches with `leave`, we simply check if the player is in the map of players in which case we remove him. Otherwise we send back an error in the loop function. Since the `leave` function is called with the players unique reference, we assume that we can safely remove the player if called.

The `guess` function send a message from a player to the quiz. In the `loop` we check to make sure that the reference passed as an argument in the `guess` function corresponds to a player in our map. In which case we store the guess in an element of the tuple of status. Otherwise we just ignore the guess.

The `timesup` function does multiple things. Like the rest of the functions, the majority of the logic is in the `loop` function. If there is an active question, the guesses are processed and the questions are dequeued. If the guess is correct a point is awarded to the respective player. Then the scores are broadcast back to the players. If the queue is empty once the question is dequeued. The quiz is finished.

2 Testing

We started by testing in the erlang shell. Here is an exampile REPL session,

```
8> {_, A} = quizmaster:start().
{ok,<0.95.0>}
9>  quizmaster:add_question(A, {"who is my partner",
[{correct, "nicolas"}, "philip", "tom"]}).
ok
10>  quizmaster:add_question(A, {"who is my cool",
[{correct, "Jason"}, "sam,", "lily"]}).
ok
11> quizmaster:get_questions(A).
[{"who is my partner",[{correct,"nicolas"},"philip","tom"]},
 {"who is my cool",[{correct,"Jason"},"sam,","lily"]}]
12>
```

We also have a test file called `test_quizmaster.erl` which we run unit tests on our different functions. We indubitably test start, get questions and add questions in conjunction as they are helpful for debugging each other. So

our test case is an amalgamation of the function calls to start, add question, and get questions. We test the other functions in the same way.

Similarly in order to test the other functions we needed to interleave their functions calls. Thus we implement a function called pre-populate which files a quiz-master process with questions similar to the ones used in our previous test. And using this pre-populated quiz master we test the remainder of the functions.

You can run the tests by starting a shell session. in erlang

```
8> c(test_quiz).
{ok,test_quiz}
9> test_quiz:test().
```

3 Assessment

Even though we were able to successfully implement quizmaster such that it met the specifications, Some of our code was a little clunky and hard to read. The best example of this, is our idea for the `Status` parameter for the loop function. This can sometimes be represented (in the most simple form) as the atom `editable`, but in other cases, it is a triple containing the conductors pid, the status of the quiz, as well as the overall scores for the players. In the case of `guess` we even save the scores of the round in the status field.
It is possible that there is a more succinct way to encode all the fields that we want to be stored. However for our purposes the way we implemented it as a triple worked relatively well. It is also possible that implementing the questions as a queue was not necessary. We could also have implemented it as a list which like queue maintains ordering.

Our implementation restricts players from joining, when the status is not "editable". We also let somebody be a conductor once the play function has been called. These design decisions were made both with the ease of implementation in mind as well as making sure it was logical for a quiz server.
Overall, we were fairly pleased with our implementation given the number of communicating proccesses involved with the quizmaster, such as the player and the conductor. However there are likely places in our code that we could improve with more concise and practical implementations like `Status` that we discussed earlier.

4 Code

```
-module(quizmaster).

%% API exports.
-export([start/0, add_question/2, get_questions/1,
play/1, next/1, timesup/1, loop/3, get_dist/2,
inc_nth/2, playerscore/1, hideAnswers/1, addToTotal/2,
player_free/2, join/2, leave/2, guess/3]).

% This is the quiz loop. Questions is a queue, pretty self-explanatory.
% Players is a map, where we decided to store the players by their reference, for easy l
% Status however, can either be the atom "editable", or some {Conductor, State, Score}
% Where state will also contain the guesses of the players, if a question is active
loop(Questions, Players, Status) ->
Me = self(),
receive
{From, {add_question, Description, MarkedAnswers}} ->
case Status of
editable ->
```

```
From ! {Me, ok},
loop(queue:snoc(Questions, {Description, MarkedAnswers}), Players, Status);
_ -> From ! {error, we_are_playing},
loop(Questions, Players, Status)
end;
{From, get_questions} ->
From ! {Me, {questions, Questions}},
loop(Questions, Players, Status);
{From, play} ->
case Status of
editable ->
From ! {Me, ok},
loop(Questions, Players,
{From, playing_between_questions, playerscore(Players)});
_ ->
From ! {Me, {error, conductor_exists}},
loop(Questions, Players, Status)
end;
{From, next} ->
case Status of
{Conductor, playing_between_questions, Scores} ->
if
From =:= Conductor ->
Question = hideAnswers(queue:get(Questions)),
From ! {Me, {ok, Question}},

% Sends next_question to player, to be used with map
SendToPlayer =
fun(Ref, {_, Pid}) ->
  Pid ! {next_question, Ref, Question}
end,

maps:map(SendToPlayer, Players),
% We store the guesses in this map
loop(Questions, Players,
{Conductor, {playing_active_question, #{}}, Scores});
true ->
From ! {error, who_are_you},
loop(Questions, Players, Status)
end;
{_, {playing_active_question, _} , _} ->
From ! {Me, {error, has_active_question}},
loop(Questions, Players, Status);
_ ->
From ! {Me, {error, not_playing_yet}},
loop(Questions, Players, Status)
```

```
end;
{From, times_up} ->
case Status of
{Conductor, {playing_active_question, Guesses}, Scores} ->
if From =:= Conductor ->
case queue:out(Questions) of
{{_, {_, Answers}}, RemQuestions} ->

% Give points for a single guess
GivePoints =
fun(_, Index) ->
case lists:nth(Index, Answers) of
    {correct, _}  -> 1;
    _             -> 0
end
end,

Dist = get_dist(Guesses, length(Answers)),
LastQ = maps:map(GivePoints, Guesses),
NewTotal = addToTotal(maps:to_list(LastQ), Scores),
Final = queue:is_empty(RemQuestions),

From ! {Me, {ok, Dist, LastQ, NewTotal, Final}},
% We still need to remove placeholders

if
% game should end
Final ->
% Sends a message to a player, to be used with map
SendToPlayer =
  fun(_, {_, Pid}) ->
    Pid ! {Me, quiz_over}
  end,
maps:map(SendToPlayer, Players),
% Just to reinitialize the quiz
loop(RemQuestions, #{}, editable);
true ->
loop(RemQuestions, Players,
    {Conductor, playing_between_questions, NewTotal})
end;
_ -> From ! {Me, {error, question_wrong_format}}
end;
true ->
From ! {Me, {error, nice_try}},
loop(Questions, Players, Status)
end;
```

```
{_, playing_between_questions, _} ->
From ! {Me, {error, no_questions_asked}},
loop(Questions, Players, Status);
_ -> From ! {Me, {error, not_even_playing}},
loop(Questions, Players, Status)
end;
{From, join, Player} ->
case Status of
editable ->
NameIsFree = player_free(Player, maps:values(Players)),
if
NameIsFree ->
Ref = make_ref(),
NewPlayers = maps:put(Ref, {Player, From}, Players),
From ! {Me, {ok, Ref}},
loop(Questions, NewPlayers, editable);
true ->
From ! {Me, {error, is_taken}},
loop(Questions, Players, Status)
end;
_ ->
From ! {Me, {error, cant_join_yet}},
loop(Questions, Players, Status)
end;
% Players does not have to be deleted from the scoreboard,
% so there's no reason to check the status
{From, leave, Ref} ->
UserExists = maps:is_key(Ref, Players),
if
UserExists ->
From ! {Ref, ok},
loop(Questions, maps:remove(Ref, Players), Status);
true ->
From ! {Ref, {error, who_are_you}},
loop(Questions, Players, Status)
end;
{_, guess, Ref, Index} ->
case Status of
{Conductor, {playing_active_question, Guesses}, Scores} ->
PlayerExists = maps:is_key(Ref, Players),
if
PlayerExists ->
loop(Questions, Players,
{Conductor,
{playing_active_question, maps:put(Ref, Index, Guesses)},
Scores});
```

```
true ->
loop(Questions, Players, Status) % We just ignore the guess
end;
_ -> loop(Questions, Players, Status) % We just ignore the guess
end;
{From, _} ->
From ! {Me, {error, "Arguments are on the wrong form"}},
loop(Questions, Players, Status)
end.

% We get the keys already defined in Players, then map each key to a {key, 0}.
% This way our scoreboard is initialized. Then we convert this new list to a map.
playerscore(Players) ->
Keys = maps:keys(Players),
maps:from_list(lists:map(fun(X) -> {X, 0} end, Keys)).

% Check if player name is free
player_free(_, []) -> true;
player_free(Player, [{Name, _} | Players]) ->
if
Player =:= Name -> false;
true -> player_free(Player, Players)
end.

% Increment the nth element of a list
inc_nth(_, []) -> [];
inc_nth(N, [X | Xs]) ->
if
N =:= 1 -> [X + 1 | Xs];
true -> [X | inc_nth(N-1, Xs)]
end.

% Builds a distribution of answers
get_dist(Guesses, NumAnswers) ->
Initial = lists:duplicate(NumAnswers, 0),
Indexes = maps:values(Guesses),
Fun = fun(X, List) -> inc_nth(X, List) end,
lists:foldl(Fun, Initial, Indexes).

% Just removes the correct part of {correct, answer}
hideAnswers({Description, Answers}) ->
HideAnswer =
fun(X) ->
case X of
{_, Answer} -> Answer;
Answer -> Answer
```

```
end
end,
{Description, lists:map(HideAnswer, Answers)}.

% Increments the value for key Pid in Total by each score in a list of {Pid, Score}
addToTotal([], Total) -> Total;
addToTotal([{Ref, Score} | Scores], Total) ->
CurrentScore = maps:get(Ref, Total, 0),
NewTotal = maps:update(Ref, Score + CurrentScore, Total),
addToTotal(Scores, NewTotal).

%Need to add a pattern match for case of failure
start() -> {ok, spawn(quizmaster, loop, [queue:new(), #{}, editable])}.

add_question(Q, {Description, MarkedAnswers}) ->
Q ! {self(), {add_question, Description, MarkedAnswers}},
receive
{Q, ok} -> ok;
{Q, Message} -> Message
end.


get_questions(Q) ->
Q ! {self(), get_questions},
receive
{Q, {questions, Questions}} -> queue:to_list(Questions)
end.

play(Q) ->
case get_questions(Q) of
[] -> {error, no_questions};
_ ->
Q ! {self(), play},
receive
{Q, ok} -> ok;
{Q, Error} -> Error
end
end.

next(Q) ->
Q ! {self(), next},
receive
{Q, Message} -> Message
end.

timesup(Q) ->
```

```
Q ! {self(), times_up},
receive
{Q, {error, nice_try}} -> nice_try;
{Q, Message} -> Message
end.

join(Q, Player) ->
Q ! {self(), join, Player},
receive
{Q, {ok, Ref}} -> {ok, Ref};
{Q, {error, Reason}} -> {error, Reason}
end.

leave(Q, Ref) ->
Q ! {self(), leave, Ref},
receive
{Ref, ok} -> ok;
{Ref, {error, Reason}} -> {error, Reason}
end.

guess(Q, Ref, Index) ->
Q ! {self(), guess, Ref, Index}.
```