Advanced Programming

Assignment 4: Kaboose

Per Steffen Czolbe, Konrad Gnoinski

Handed in: October 10, 2017

Contents

Description of the Assignment

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]

Solution

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

Tools

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.
  - `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 in the eshell.

Running Tests

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]).
```

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 uestions to a room and spawning 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 asyncronous communication pattern was added.

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. It is not specified when a game room should terminate. Do they just keep being around forever?
3. 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.

Assessment

This section contains the assessment of the presented solution.

Scope of Test Cases

The `kaboose_tests.erl` file contains ...

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.