Dr. Annette Bieniusa
Albert Schimpf, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Exercise 4: Programming Distributed Systems (Summer 2020)

Submission

- You need a team and a Gitlab repository for this exercise sheet.
- In your Git repository, create a branch for this exercise sheet (for example with `git checkout -b ex4`)
- Create a folder named "ex4" in your repository and add your solutions to this folder.
- Create a merge request in Gitlab and assign Albert Schimpf as assignee. If you do not want to get feedback on your solution, you can merge it by yourself.
- Test your submission with the provided test cases. Feel free to add more tests, but do not change the existing test cases.

1 Vectorclock Service

Implement a vectorclock `gen_server` named `vc_server` based on your implementation of a vectorclock in Erlang (from Exercise Sheet 3). It should provide the following API:

- `start_link()`: Starts the server.
- `stop()`: Stops the server.
- `tick(P)`: Increment the current vectorclock at the entry for process `Pid`.
- `advance_to(VC)`: Merge the current vectorclock with vectorclock `VC` given as a list.
- `get(P)`: Returns the current vectorclock entry for process `Pid`.
- `get_vc()`: Returns the current vectorclock as list.

Remarks:

- To encapsulate the vector clock implementation, vector clocks will be passed between clients and the server as tuple lists.
- You can use the `vc_server_tests` to test your implementation; to this end, the server must be locally registered under the name `vc_server`.

2 Chat Service

Download the supplied materials `chatty.zip` from the website. It contains two rebar3 projects, one for a chat server and one for a chat client.

- To start a server, execute `rebar3 shell --sname chatty@localhost` in the `chatty` folder.
- To start any client X and connect to the server:
  1. Go to the `chatty_client` folder
  2. Execute `rebar3 shell --sname X@localhost`, where X is the client name
  3. Connect to the server node with `net_adm:ping(chatty@localhost)` inside the shell

A simple chat client is implemented and ready for use. The chat client knows how to communicate with the global process `chatty` (after connecting to the server) and you can use the following client API

- `chatty_client:join()`: Add yourself to the chat server

- `chatty_client:leave()`: Remove yourself from the chat server
- `chatty_client:message(X)`: Send the message X to the chat server

Your task is to finish the implementation of the chat server.

2.1 Join and Leave

Implement `handle_call` callbacks for the chat server requests `{action, join}` and `{action, leave}`. The in-memory state of the `gen_server` can be used to remember client PIDs.

2.2 Message

Implement a `handle_call` callback for the chat server request `{message, Message}`. Whenever a client sends this request, the server is supposed to send that message to all known chat clients currently connected.

2.3 Improve Server or Client (Optional)

Improve the chat server or the chat client one with the following aspects:

- Modify the protocol to include nicknames when joining a chat server
- Persist the client list of the chat server, such that after a restart or crash the server works as expected
- Clients receive a history of the last X messages sent

... or design your own features!