Prof. Dr. Annette Bieniusa
Philipp Lersch, M. Sc.

RPTU Kaiserslautern-Landau
Fachbereich Informatik
AG Softwaretechnik

Exercise 5: Programming Distributed Systems (Summer 2025)

Deadline: 28.5.25 AOE

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 ex5`)
- Create a folder named "ex5" in your repository and add your solutions to this folder.
- As always assign me (Philipp Lersch) to a merge request to receive feedback.

1 Chat Service

Tipps on how to connect distributed nodes:

- To start an Elixir node, execute `iex --sname node_name@localhost --cookie secret -S mix` in the `chatty` folder. `node_name` must be unique for every node joining a cluster or an error will be returned
- The Node module has additional useful functionality:
  1. Check if a node is alive `Node.alive?/0`
  2. Connect to the server node with `Node.ping(:"chatty1@localhost")` inside the shell. Notice how Node names are written as atoms.
  3. `Node.self/0` returns the node's name (Different to `self?/0` for the PID)
  4. `Node.list/0` returns all reachable nodes.
- When starting a GenServer there are useful features to ease referencing available instances:
  1. A GenServer can node-wide be named by passing a thrid argument:
     `GenServer.start_link(GenServerImpl, inital_state_parameter, name: :TheNewNameForReference)`
  2. A GenServer can also be globally for the whole cluster be named by changing the thrid argument:
     `GenServer.start_link(GenServerImpl, inital_state_parameter, pid = {:global, :TheNewNameForReference})`
  3. These names can then be dereferenced to PIDs via
     `Process.whereis(:TheNewNameForReference)`
- Finally it is possible to have process groups through erlangs `:pg` module, which provide a way to register and query pids on different nodes:
  1. Connect to the process: `:pg.start_link()`
  2. Register a pid to a group: `:pg.join(:group_name, self())`
  3. Request pids by group: `:pg.get_members(:group_name)`
- Feel free to use any Elixir mechanism besides the above listed as you see fit

The goal of this exercise is to use the distributed message passing mechanism of Elixir to design and implement a chat service.

1.1 Architecture

Plan and draw an architecture diagram of your chat service.

In the example project that is provided with the materials an example architecture skeleton is provided for you. Feel free to implement your own code from scratch.

In the template, the user-facing API provides the following functions:

- Joining: Add yourself to a chat and its list of participants
- Leaving: Remove yourself from the chat
- Send a message: Send a message to the chat

Two possible options for an architecture are:

1. A globally named GenServer that handles all messages send by clients
2. Per Beam VM named GenServers that handle the interactions from the clients on the same BEAM VM and coordinate with other GenServers on other BEAM VMs. These can be made known through the :pg module.

The architecture diagram should include at least the following:

- Phsyical nodes; possibly many BEAM VMs will be running on one physical node
- Virtual nodes; one chatty app is executed on one virtual node. Possibly many virtual nodes can be mapped to physical nodes. One virtual node is assigned to at most one physical node.
- (Elixir) processes which are running inside one virtual node
- What and why communication between which processes happens

Also think about the following questions, while keeping in mind your broadcast implementations:

- How do the Elixir chat processes communicate? Message passing directly (global name register), using the network layer of your broadcast algorithm?
- (single) Server-client architecture?
  - Are the server and client apps mapped to a physical or to a virtual node?
  - Are server and client always bundled and running on the same physical node? Or can they be completely separated?
- Is peer to peer architecture possible? What about multiple servers?

You can assume the environment of the chatty application will be run in a trusted local network where every node can be fully connected in a distributed Elixir mesh.

Shortly describe the implications of running your application in an untrusted local or globally accessible network. You can assume that a distributed Elixir node mesh can be correctly established in such an environment.

1.2 Gen Server - Chat

Implement a chat based on your proposed architecture, it should have at least a join, leave, and messaging mechanism.

1.3 Improve The Chat (Optional)

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

- Modify the protocol to include nicknames when joining a chat
- Allow direct messages that only appear at the targeted client
- Clients receive a history of the last X messages sent when the join a group
- Allow multiple rooms with independent chats
- Improve resilience by adding a reliable broadcast (probably requires a multi server configuration)
- List all rooms and their participants
- Allow to kick clients
- ... be creative and design your own features!