Software Architectures

Assignment 3: Actor-based design patterns

Camilo Velazquez, Ahmed Zerouali
Emails: {cavelazq, azeroual}@vub.be
Office: {10F725}

For this assignment, you will need to implement a micro-service architecture using actor-based design patterns.

Assignment

For this assignment you will implement and report on your original solution to an architectural problem.

Deadline: January 10^{th} 2021 by 23:59. The deadline is fixed and will not be extended for any reason.

Deliverables A report and source code. The report (in English) explains your solution to the problem and the main components used in your implementation. When possible, use simple diagrams to illustrate.

The report should be handed in as a single PDF file of no more than 3 pages (excluding diagrams/screenshots/code). The file should follow the naming schema `FirstName-LastName-SA3.pdf`, for example: `Camilo-Velazquez-SA3.pdf`.

The source code is your implementation of the project zipped or exported from IntelliJ.
Submit the *report* and *source code* as a single zip file on the Software Architectures course page in Canvas, by clicking on *Assignments* > *Assignment 3*.

Plagiarism Note that copying – whether from previous years, from other students, or from the internet – will not be tolerated, and will be reported to the dean as plagiarism, who will decide appropriate disciplinary action (e.g., expulsion or exclusion from the examination session). If you use any other resources besides those provided in the lectures and in this document, remember to cite them in your report.

Grading Your solution will be graded and can become subject of an additional defense upon your or the assistants' request.

Problem Description

For this assignment, you will design and implement an online shopping store based on a micro-service architecture. Products in the store can be bought and delivered to the customer's address. The setup will contain a set of necessary objects. Each object ending with `Service` should be implemented as an actor. `Services` should take care of the communication logic while other objects should take care of the business logic. The set of objects is defined as follows:

- A `Product` is identified by its name.
- A `Purchase` is identified by a collection of products and their quantity.
- A `ClientService` will handle the communication on behalf of the `Client`. The latter is identified by its name and address[1]. The `ClientService` will send the purchases to the `ProcessingService`.
- A `ProcessingService` will process the products requested by the `ClientService`.
- A `StockHouseService` will manage a `StockHouse` and represent its communication logic. The `StockHouseService` will take care of checking and sending product availability in the stock when requested from the `ProcessingService`.
- A `StockHouse` will encapsulate the domain logic of the stocks by including a set of products and their quantity. The `StockHouse` will also have an address (i.e., `Point`) assisting distance measuring with respect to a `Client`.

A `ClientService` can buy `Product`s by including them in a `Purchase` list. The confirmation of the purchase is made through the use of a `PurchaseConfirmed` message from the `ClientService` to the `ProcessingService`.

The `ProcessingService` will then collect all products selected and placed in the purchase. In order to do so, it will need to contact the `StockHouseService`s to check the products availability in the 3 nearest `StockHouse`s. The closest `StockHouse`s to the client can be calculated using the following formula: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Once the nearest stock houses are identified, the `ProcessingService` will query each of them for the availability of the products via a message `FillOrder`. Such a message will contain the name and quantity of every product requested by the `Client`. The `StockHouseService` needs to be concerned with the communication logic while the `StockHouse` should only be concerned with business logic. These two concerns need to be clearly separated in the implementation.

`InStock` messages with the names of available products and their quantity will be gathered by the `ProcessingService` (i.e., the `ProcessingService` aggregates responses from the `StockHouseService` with the closest `StockHouse`, stops aggregating when sufficient responses have arrived to fill the order, etc). Once all products are ready, an

[1]For testing purposes the address should be defined as a `Point` with cartesian coordinates.

`OrderShipped` message needs to be sent back to the same `Client` requesting the products. However, if an order is not yet complete due to delays in the stock houses or lack of available products, an `OrderDelayed` message have to be sent instead.

In addition to the general approach, you will be evaluated according to how you implement your actor-based architecture. You should choose patterns studied in class for your implementation of the solution. For example, you should select at least two patterns from the following list: 1) Forward Flow 2) Aggregator 3) Domain Object 4) Business-Handshake and 5) Managed Queue.

You can also take into account other abstractions from the lectures that you deem fit for the scenario. **Motivate your design decisions** in the report as per the problem description and illustrate your approach using concepts from Akka actors and actor-based design patterns.

More information on Actors and Akka

- `https://doc.akka.io/docs/akka/2.5.32/`
- `https://doc.akka.io/docs/akka/2.5.32/index-actors.html`