ALCHEMIST SIMULATION BATCH DISTRIBUTION

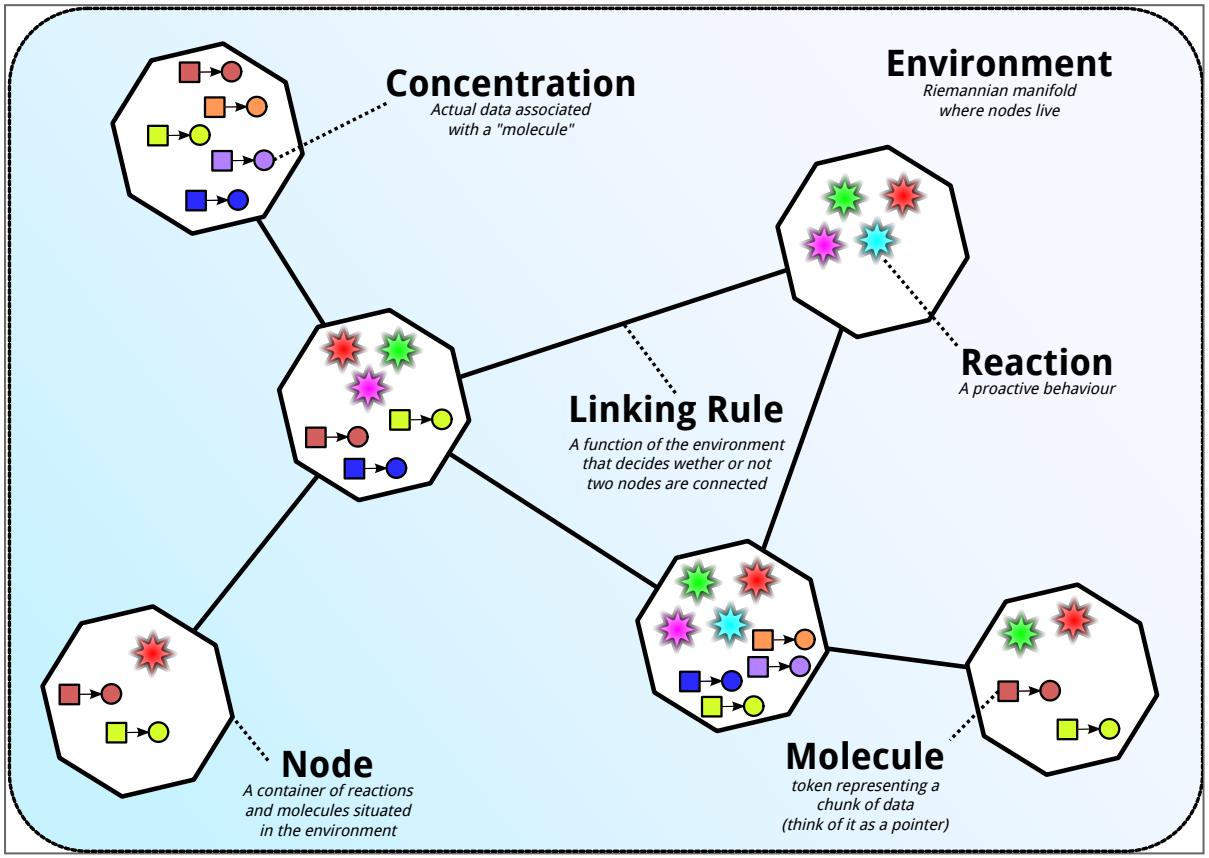
DISTRIBUTED SYSTEMS PROJECT

Kelvin Oluwada Milare Obuneme Olaiya
kelvinoluwada.olaiya@studio.unibo.it
A.Y. 2022/2023
report

INTRODUCTION

ALCHEMIST

- Is a chemical-oriented general-purpose simulator
- Open source



Alchemist’s meta-model

INTRODUCTION

ALCHEMIST

- To run one simulation one should:
  1. Write a simulation configuration file in YAML
  2. Launch the simulator
  3. Wait for the completion
  4. Possibly analyze any exported data

THE PROBLEM

- Sometimes it may be useful to execute the ***same configuration*** with ***different parameters***, called **variables**.
- The set of simulation differing by their variables constitute a **batch**.
- Alchemist provides a way to launch a simulation batch ***sequentially***.

> Running a simulation can be time-consuming, let alone running a simulation batch.

THE PROBLEM

THE NEED OF DISTRIBUTION

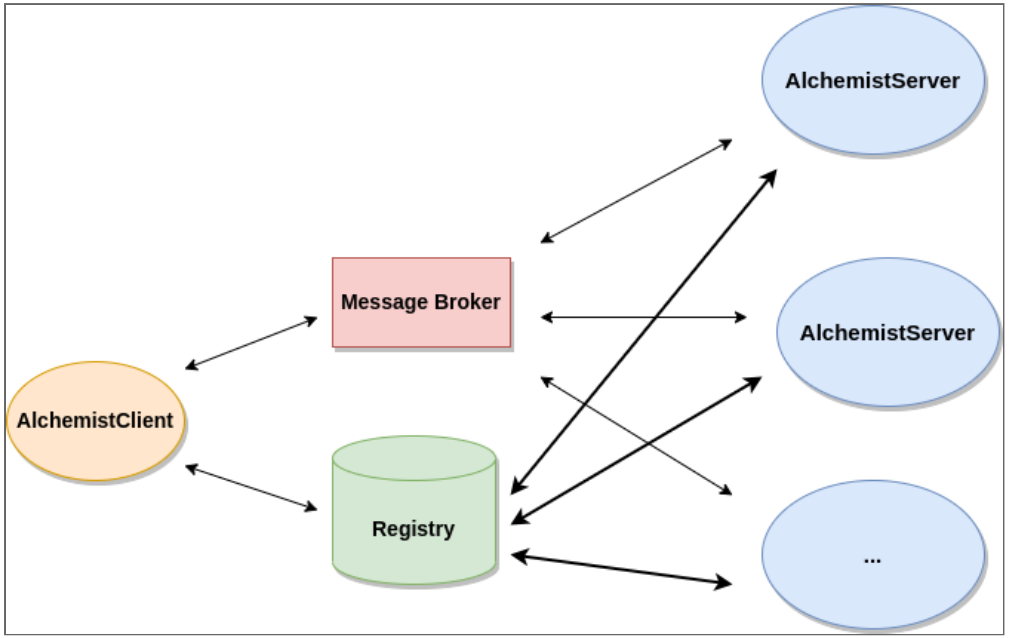
Taking advantage of ***multiple computing resources*** can be a way to ***reduce the time*** necessary to complete the execution of a simulation batch.

FUNCTIONAL REQUIREMENTS

- It should be possible to create a **cluster of nodes**, each executing a service exposing Alchemist.
- Alchemist should provide a way to **distribute a batch of simulations** to be executed by one or more nodes on a cluster.
- Each node of a cluster must be up and ready to receive and execute configurations of simulations.
- **None of the distributed simulations should get lost**, meaning that in case of a node failure, a **recovery** mechanism should redistribute the simulations assigned to the failing node.
- Once a simulation is computed by a node, results should be made available to the user who launched the distribution.

DESIGN

ARCHITECTURE

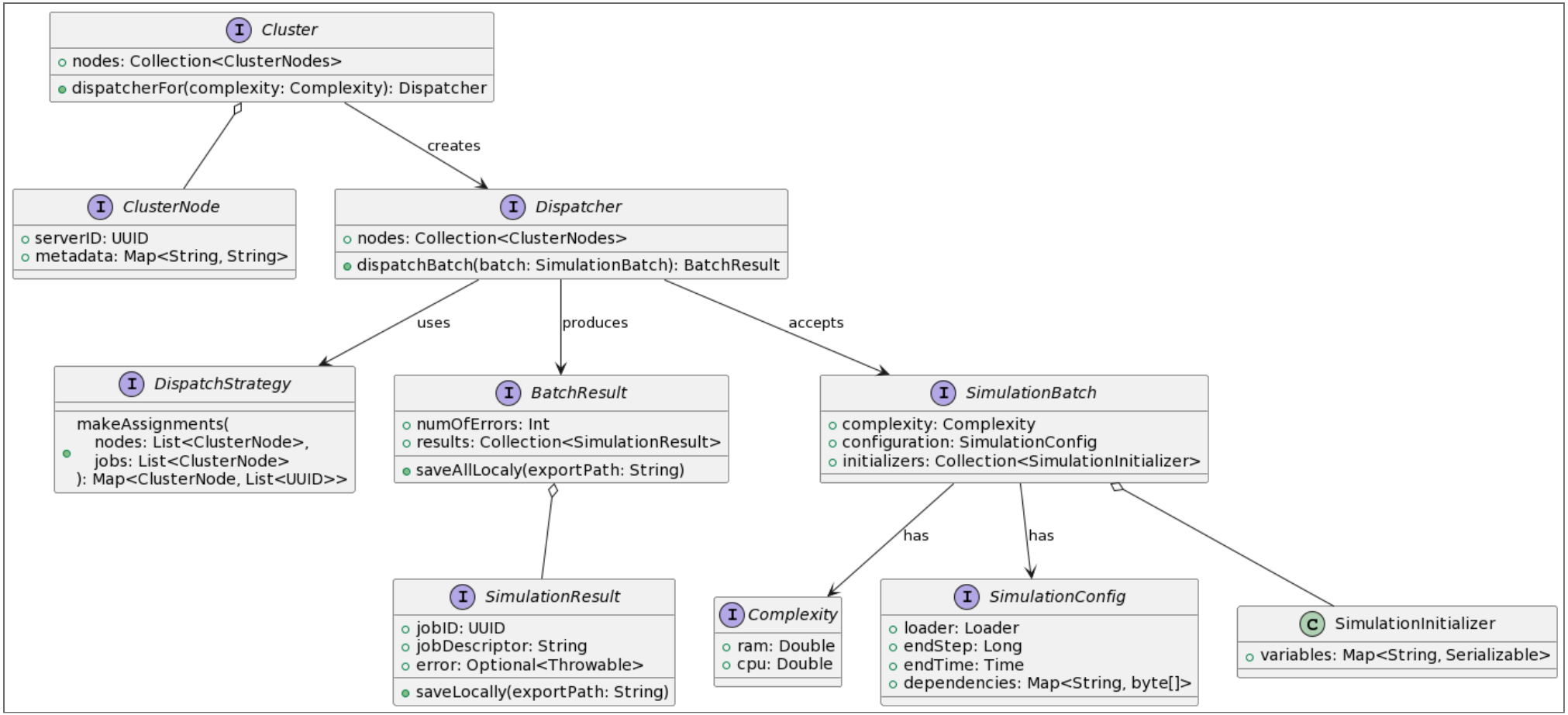


Main actors:

- **AlchemistClient**: Loads and distributes a simulation batch.
- **AlchemistServer**: Waits for job orders (mainly simulations) to execute.
- **Registry**: management of all the information that guarantees the correct functioning of the system

- **Message Broker**: responsible for the communication between nodes in the cluster.

DOMAIN STRUCTURE:



DOMAIN STRUCTURE (1/3)

- **Cluster** is an entity representing the collection of nodes that are currently connected forming a cluster. Through the cluster it is possible to obtain a ***Dispatcher***, specifying the complexity that the nodes in the dispatcher should be able to handle.
- **ClusterNode** represent a server node to which jobs can be distributed.
- **Dispatcher** contains a subset of the nodes in the cluster. It is responsible for accepting ***SimulationBatches*** and distribute them across subset of nodes. Distribution is made according to a ***DispatchStrategy***
- **DispatchStrategy** it models the strategy with which the work load gets distributed to a collection of nodes (e.g. *round-robin*).

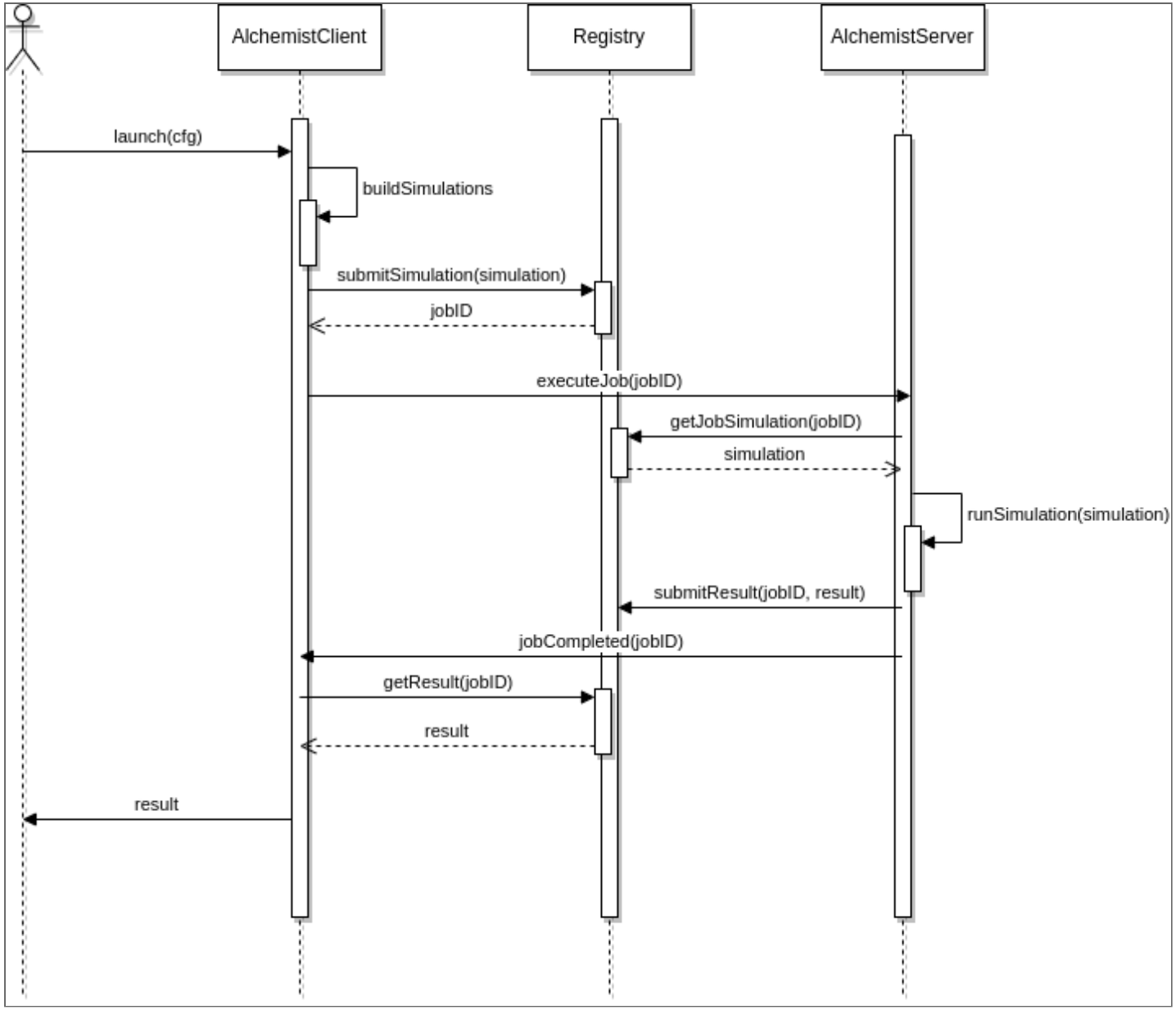
DOMAIN STRUCTURE (2/3)

- **Complexity** describes the complexity in terms of ram usage and memory occupation for the simulations in a batch.
- **SimulationBatch** represents a simulation batch with its complexity. It is composed of a simulation configuration and a collection of simulation initializers.
- **SimulationConfig** contains the general batch information such as the end step and end time of the simulations and a loader from which simulation instances will be created. ***Dependencies*** are files that must be made available to all servers in order to execute the simulation correctly.

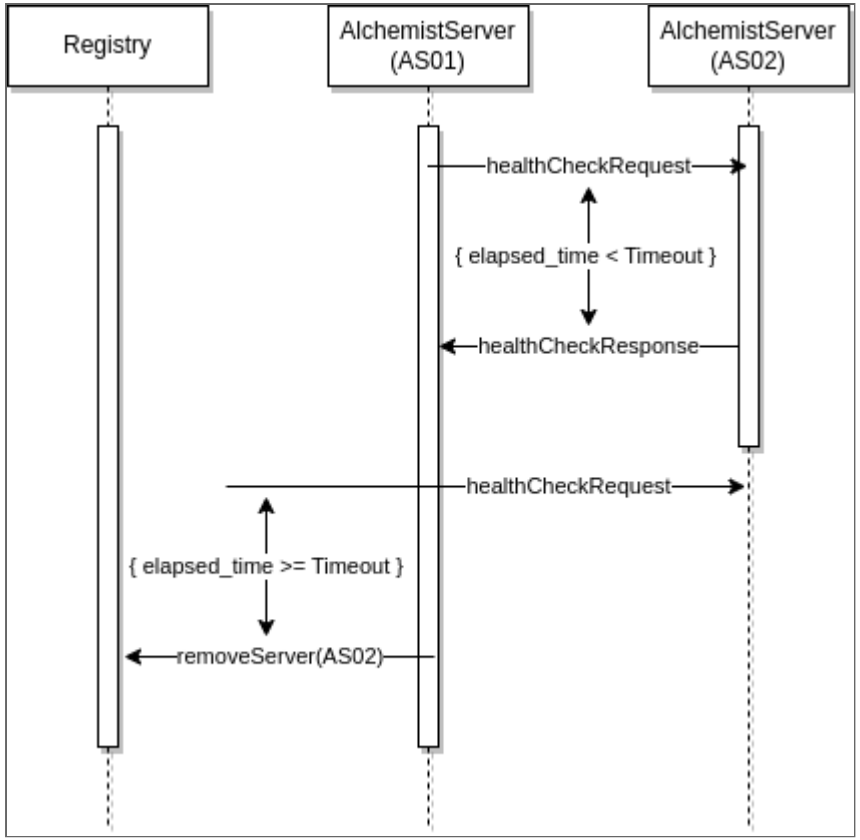
DOMAIN STRUCTURE (3/3)

- **SimulationInitializer** contains a combination of variables values that will be used to create a simulation instance. For every simulation initializer in a simulation batch corresponds a job for a node in the cluster.
- **BatchResult** models the result of a simulation batch that have been submitted via a ***Dispatcher***. It gives information on the total number of errors, if any, that have occurred while executing the simulation batch and a utility method to save all the distributed export files locally.
- **SimulationResult** models the result of a single job.

MAIN INTERACTIONS

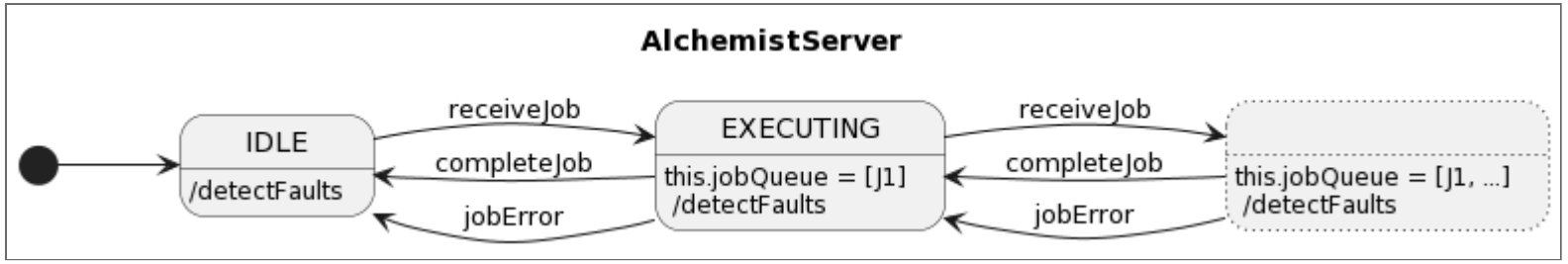


Simulation distribution

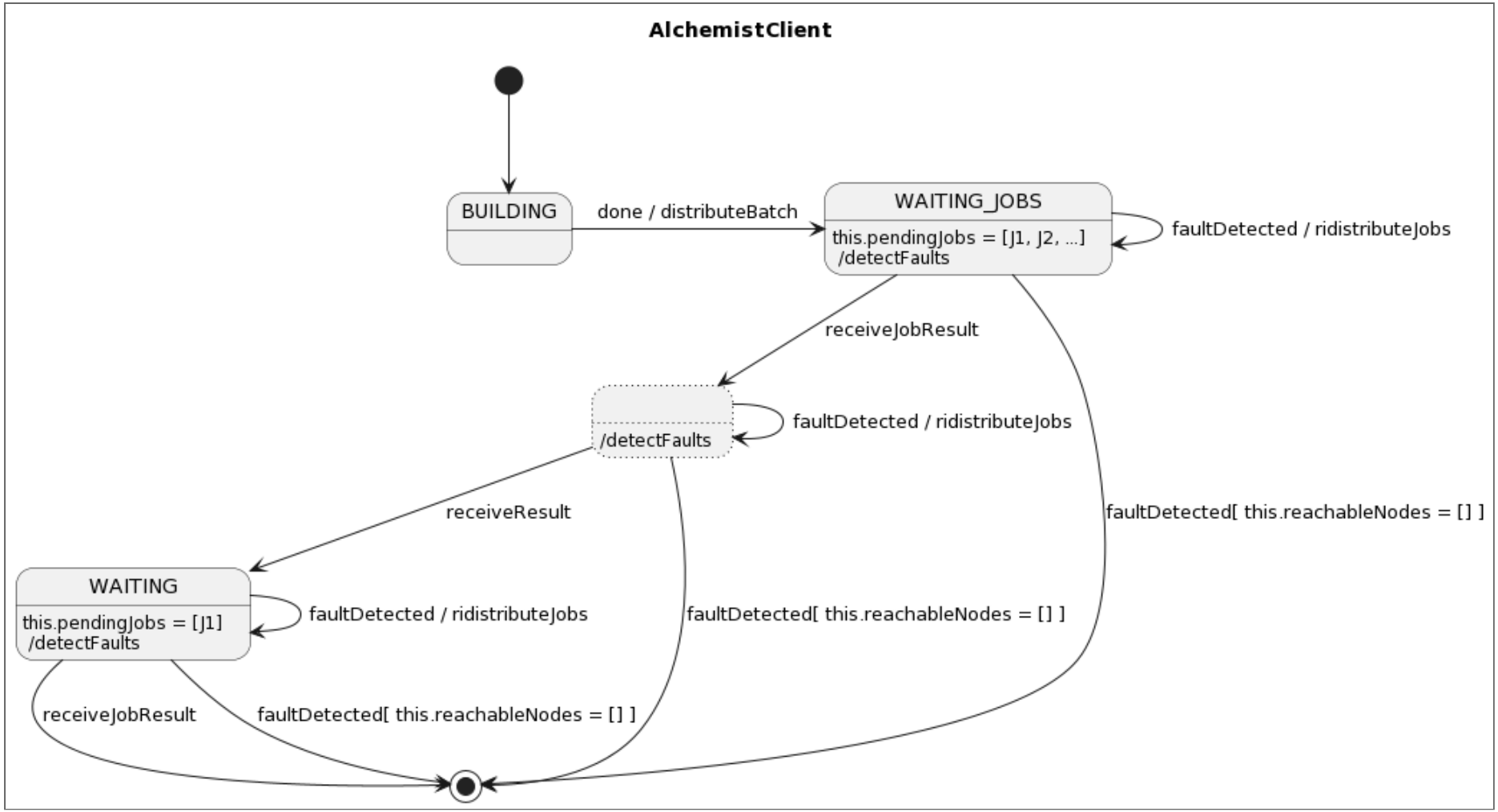


Fault detector

BEHAVIOR



Alchemist server

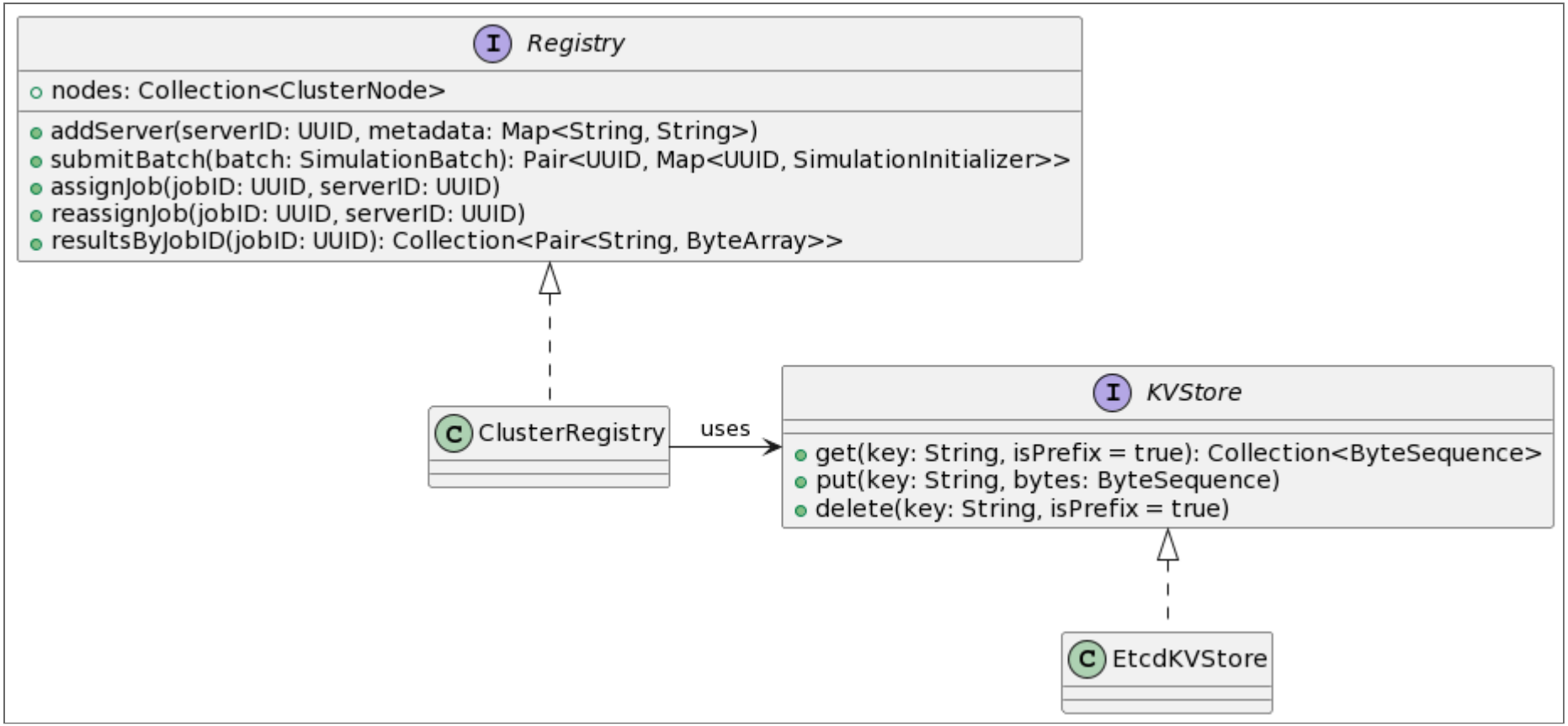


Alchemist client

IMPLEMENTATION DETAILS

TECHNOLOGIES

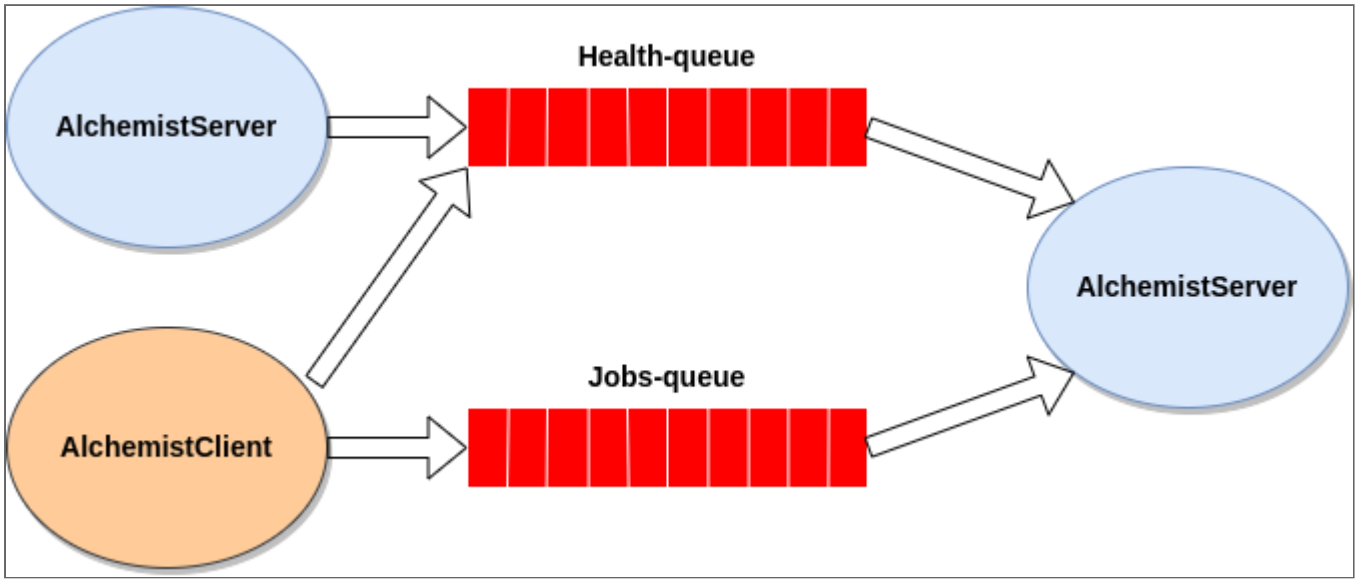
- **Etcd**: a distributed, reliable and strongly consistent key-value store. It has been used to store the most important data for the functioning of the system (The registry)


Registry
nodes: Collection<ClusterNode>
addServer(serverID: UUID, metadata: Map<String, String>)
submitBatch(batch: SimulationBatch): Pair<UUID, Map<UUID, SimulationInitializer>>
assignJob(jobID: UUID, serverID: UUID)
reassignJob(jobID: UUID, serverID: UUID)
resultsByJobID(jobID: UUID): Collection<Pair<String, ByteArray>>
ClusterRegistry
uses
KVStore
get(key: String, isPrefix = true): Collection<ByteSequence>
put(key: String, bytes: ByteSequence)
delete(key: String, isPrefix = true)
EtcdKVStore

IMPLEMENTATION DETAILS

TECHNOLOGIES

- **RabbitMQ**: an open-source message broker based on the Advanced Message Queuing Protocol (AMQP) for reliable communication.



IMPLEMENTATION DETAILS

TECHNOLOGIES

- **Protobuf**: Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.

```
message Simulation {
  string simulationID = 1;
  bytes environment = 2;
  bytes exports = 3;
  string jobDescriptor = 4;
}
```

TESTING

- A series of test have been written to assess whether the system complies with the project requirements.
- Main challenges (as with distributed systems in general): dealing with **asynchronous behavior** and **non-determinism**.
- For this the Kotest testing framework came in handy.

TESTING

A test example:

```
"Simulation are correctly distributed" {
        startServers(serverConfigFile, SERVERS_TO_LAUNCH).use {
            val cluster = ClusterImpl(registry)
            awaitServerJoin(cluster, SERVERS_TO_LAUNCH, 10.seconds)
            startClient(clientConfigFile).use {
                until(20.seconds) {
                    registry.simulations().size == 1
                }
                val simulationID = registry.simulations().first()
                registry.simulationJobs(simulationID) shouldHaveSize SIMULATION_BATCH_SIZE
            }
        }
    }
```

Kotest functions for non-determinism:

- `eventually`
- `continually`
- `until`

FUTURE WORKS

- Implementation of a cluster monitoring & management **dashboard**
- Improvement on the **fault detection routine**
- New dispacth strategies based also on the **heterogeneity of the computing nodes** and the **complexity of a simulation batch**

DEMO

repository

Speaker notes