Fog Computing - Agricultural Scenario

Documentation

Jonathan Kossick & Konstantin Köhler

Scenario

The Client is an agriculture bot which is able to do two different tasks: Watering and Inspecting plants (checking soil humidity and pH level) on a farm. Both tasks will hereinafter be called missions. The client is regularly sending requests to the server to receive new missions. This request also contains information about completed missions including simulated sensor data (soil humidity and pH level).

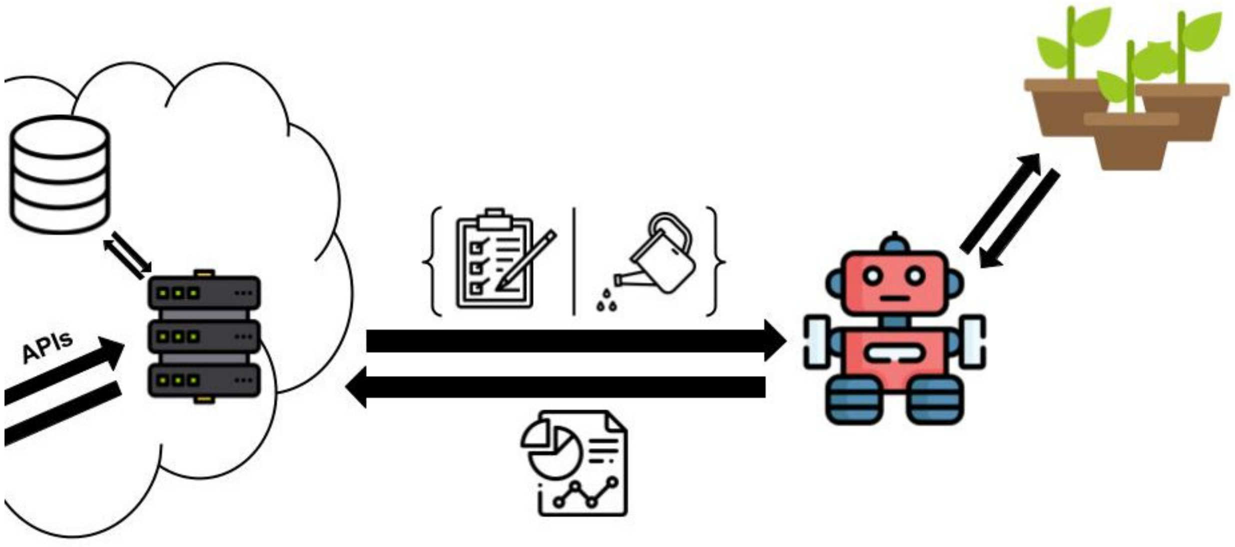

Figure 1: Agricultural Scenario

The server is responsible to supply new missions to the bot and to save the results of completed missions permanently.

Further logic like the use of sensor data to determine when a plant should be watered next is neglected in the actual implementation as this is only a scenario to exemplify a fog application which has to cope with fog-specific challanges. In a real life scenario, the server would use the inspection data and inforamtion about the weather to calculate the amount of water, that the respective plant ideally needs to be watered with.

Technology

Client and server are written in Kotlin. Both share common code like the database access logic and strongly typed serializable data classes. Messages are serialized and deserialized using *kotlinx.serialization*. The respective data classes are shown in Figure 2 and Figure 3.

This project uses the simple ZeroMQ REQ/REP message pattern as the scenarios requirements are not challanging and hence do not require a more sophisticated messaging aproach. This is mainly because only a very limited amount of data needs to be transfered with time gaps in between. This data is not real-time sensitive.

For data persistance the document-based MongoDB is used. While the client database is running in a docker container, the server database runs on a MongoDB Atlas cloud service instance.

The kotlin-server is hosted by an AWS EC2 instance.

Architecture

In order to cope with fog-specific challanges as connection loss due to reception loss or hardware failure, the architecture focuses on resilience. The client application uses a local MongoDB database to buffer incoming missions which are therefore available if the connection to the server was lost (cf. Figure 4). Data resulting from completed missions are also buffered in the database. As soon as the connection to the server is restored the client sends the resulting data to the server. This will also work if the client was shut down and turned on again.

On the server side such connection lost is expected as well. For each mission which was sent to the client an expiration timestamp is saved. If the server does not hear from the client until after the expiration timestamp the server will reset that mission and send it to the next client available. This will ensure that all missions will be completed eventually, even if a client was shut down ultimately.

```kotlin
@Serializable
sealed class Mission() {
    abstract val timestamp: Long
    abstract val plant: Plant
    abstract val _id: String
    abstract var resultData: MissionResultData?
    abstract var processingExpirationDate: Long?
}

@Serializable
data class InspectionMission(
    override val timestamp: Long,
    override val plant: Plant,
    override val _id: String = newId<Mission>().toString(),
    override var resultData: MissionResultData? = null,
    override var processingExpirationDate: Long? = null,
) :
    Mission()

@Serializable
data class WateringMission(
    override val timestamp: Long,
    override val plant: Plant,
    val quantity: Int,
    override val _id: String = newId<Mission>().toString(),
    override var resultData: MissionResultData? = null,
    override var processingExpirationDate: Long? = null,
) : Mission()
```

Figure 2: Serializable data classes for mission data

```kotlin
@Serializable
sealed class MissionResultData() {
    abstract val timestamp: Long
}

@Serializable
class InspectionResultData(
    override val timestamp: Long,
    val pHLevel: Float,
    val soilMoisture: Float
) : MissionResultData()

@Serializable
class WateringResultData(
    override val timestamp: Long,
    val success: Boolean
) : MissionResultData()
```

Figure 3: Serializable data classes for result data

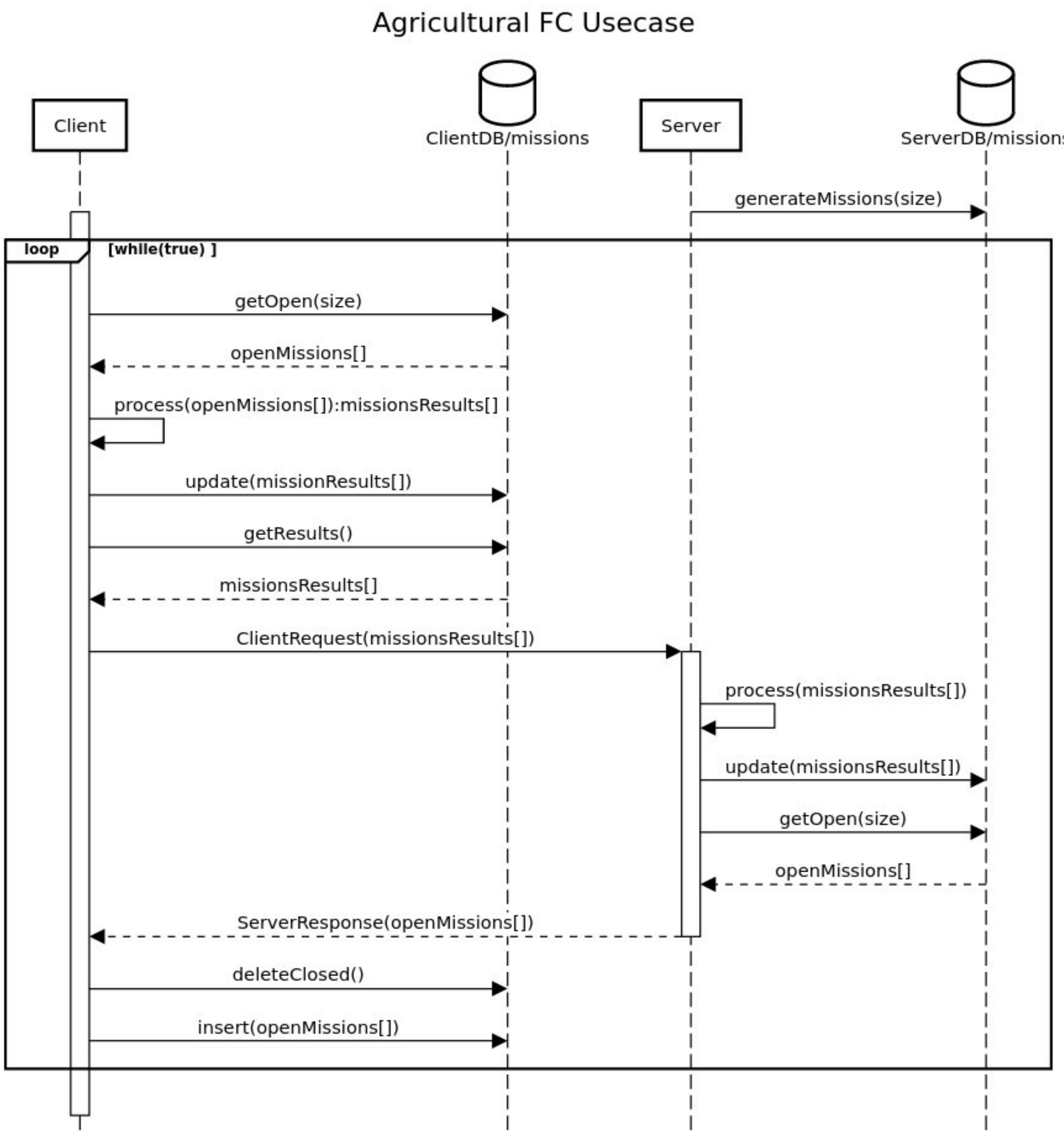

Figure 4: Sequence Diagram of Scenario