Mobility Monitoring and Simulation System

COSN

GROUP 2

João Maia	up202001526
Ana Chaves	up202003303
Hugo Andrade	up201003007

Table of Contents

Introduction

This project aims to develop a “Mobility Monitoring and Simulation System”. The architecture of this application is cloud and service oriented. It is a complex system that encompasses several different systems, and the proposed architecture is shown in Figure 1.

The focus of the Group 2 is the **Real and Artificial systems**. These systems will be service-oriented. The development process of these services will be independent from the development of the remaining systems of the project, and it will have clear API and communication protocols so that the integration in the whole project is seemingless and easy. Finally, scalability and availability tests will also be performed, in order to assess the robustness of our solution.

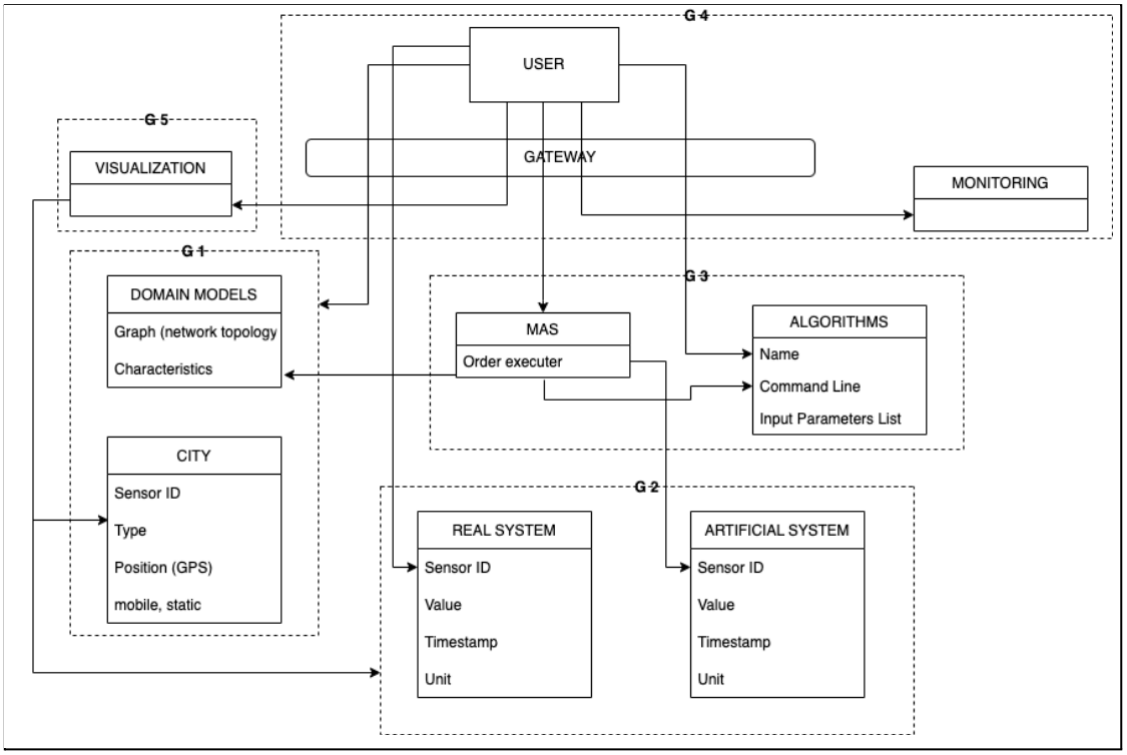

Figure 1. Scheme of the systems of the project “Mobility Monitoring and Simulation System”

Application requirements

In this section, the requirements of both systems are detailed as User Stories, each in its own subsection.

Our services have a common requirement that is the notification of the Monitoring service.

Identifier	U01
Name	Notify Monitoring
Description	Notifies Monitoring Service when any operation is executed
Actor	App developers
Preconditions	An operation is executed
Post-Conditions	Monitoring system receives the notification

Artificial System

The Actors are the researchers, since the Artificial System simulates experiences requested by them, particularly, through the MAS system.

Identifier	U02
Name	New Experience
Description	Creates a new Experience and then executes the task received with the experience.
Actor	Researchers
Preconditions	MAS sending the data and the command to be executed.
Post-Conditions	Task is executed. Task and experience are saved on the database. MAS is notified that the task was finished.

Identifier	U03
Name	Execute task
Description	Executes the task received.
Actor	Researchers
Preconditions	MAS sending the data and the command to be executed.
Post-Conditions	Task is executed. Task is saved on the database.

	MAS is notified that the task was finished.

Identifier	U04
Name	Send experiences file name.
Description	Sends a list of tasks file name to MAS system
Actor	Researchers
Preconditions	MAS system requests data. Experience with the parameters requested exists.
Post-Conditions	Experience data is returned.

Identifier	U05
Name	Send experiences data
Description	Sends experiences content to MAS system
Actor	Researchers
Preconditions	MAS system requests data. Experience with the parameters requested exists.
Post-Conditions	Experience content is returned.

Real System

The Actors are the Service Providers, which register the values of the sensors spread around the cities, and other subsystems of the whole application, which request the data that is then displayed to the Users.

Identifier	U06
Name	Update Sensor Data
Description	Receives sensor data that should be persisted.
Actor	Service Providers
Preconditions	The Sensor should be previously registered The Service Provider should be trustworthy
Post-Conditions	The Sensor Data is stored on the database A notification to the Monitoring system should be sent

Identifier	U07
Name	Return Sensor Data
Description	Returns the sensor data that is persisted
Actor	Subsystem (Visualization system)
Preconditions	The request sensors must exist
Post-Conditions	A notification to the Monitoring system should be sent

System operations

In this section, the systems operations are detailed. They derive from the requirements documented in the previous section. Again, we have decided to split both systems in subsections. However, since the user story U01 is a common requirement, it's described before the subsections.

- U01: a *command* system operator that notifies the monitoring when anything happens

Operation	NotifyMonitoringSystem(action)
Returns	-
Preconditions	An action was executed.
Post-Conditions	-

Artificial System

This system will only have two systems operations, which are directly related to the user stories U02 and U03

- U02: a *command* system operator creates a new experience and executes the task received from the MAS system, and then saves the result on the database.

Operation	newExperience(algorithm, domainModel, experienceID, fileName)
Returns	The task(timestamp, value, unit, experience, file_name) that was executed with a new experience.
Preconditions	MAS sending the data and the command to be executed.
Post-Conditions	Task is executed. Task and experience are saved on the database. A file with the processed information is created. A notification is sent to MAS using Kafka.

- U03: a *command* system operator executes the task received from the MAS system and then saves the result on the database.

Operation	executeTask(algorithm, domainModel, experienceID, fileName)
Returns	The task(timestamp, value, unit, experience, file_name) that was executed.
Preconditions	MAS sending the data and the command to be executed.
Post-Conditions	Task is executed. Task is saved on the database. A file with the processed information is created. A notification is sent to MAS using Kafka.

- U04: a *query* system operation that returns the experiences results to the MAS system

Operation	getAllTasks(experience)
Returns	A list of tasks(fileName) belonging to the experience requested.
Preconditions	MAS system requests data. Experience with the parameters requested exists.
Post-Conditions	List of tasks is returned.

- U05: a *query* system operation that returns the experiences content to the MAS system

Operation	getExperienceContent(experience)
Returns	A file with all the tasks information belonging to the experience.
Preconditions	MAS system requests data. Experience with the parameters requested exists.
Post-Conditions	FIle content is returned.

Real System

Even though this system will have a significant usage rate, it will have only two system operations, which are directly related to the user stories U06 and U07:

- U06: a *command* system operation that updates the sensor data, whose operation would be

Operation	updateData(sensorID, value, timestamp, unit)
Returns	-
Preconditions	The sensor exists
Post-Conditions	The data is stored in the database

- U07: a *query* system operation that returns the sensor data requested, whose operation would be

Operation	getData(fromTimestamp, sensorIDs...)
Returns	The sensor data(timestamp, value, unit) of the sensors whose values were updated after the provided timestamp
Preconditions	The sensor exists The fromTimestamp is in the past
Post-Conditions	-

Services

We choose to divide our system into **two services**. Our reasoning for that choice is that the Real and the Artificial systems not only don't have anything in common, they also have completely different functions. They should be autonomous, and they have completely independent business functionalities.

The Real and Artificial Systems are completely independent from each other not only in production but also during development, and therefore they will work in parallel. By having them in different services, we do not need to deploy an entire application with both services in the instances when only one of them needs to be updated.

In addition, the data they receive and store is independent and originates from different contexts, therefore they should be stored in different database instances. Therefore, splitting in two separate services, each associated with its own database, would also be beneficial to have a better organized architecture. Two database instances could be associated to an unique service that would include both systems. However, the organization would not be as clear. In case the Artificial system needs data from the Real system, it would request it from the API that is exposed from that service, just like any other services of the system.

The scalability is also a divergent issue regarding both services. They both respond to requests from Users, even if indirectly, but while the Real System responds to regular Users and other services with the latest data and with a heavy traffic, the Artificial System will be used in the context of Research, therefore it will have significantly less workload. Additionally, the need to scale up the Real System to different cities and regions of the world might imply that it must be further split into microservices (eg. by region). This requirement does not apply to the Artificial System.

Finally, by adopting this architecture we can increase resilience of each service and of the whole system. One of the systems will not be down because of a failure of the other, for reasons completely unrelated.

Service APIs and collaborations

According to the system operations previously described and the decision to have each system in its own service, we defined their Service APIs and collaborations.

The service for the Artificial System uses a message broker and REST API to communicate with the other systems. Firstly, a *command* operation “executeTask" that receives the Task that the system must execute. The system subscribes to a topic and when it has tasks to execute, it executes them, and once finished, a notification is sent to the MAS service. Secondly, a *command* operation, “newExperience”, with the same behavior as the last command operation, with the only difference being that a new experience is created before the task execution. Thirdly, a *query* system operation “getAllTasks”, invoked by the MAS System to get the Experience’s tasks files name. Lastly, a *query* system operation “getExperienceContent”, invoked by the MAS System to get the Experience’s content.

The service for the Real System exposes a REST API with two methods. Firstly a *command* query “updateData“ invoked by the service providers to update the data from the sensors. This operation needs the collaboration of the City System to verify that the sensor is valid, and this would be achieved through a REST API call. Secondly, a *query* operation “getData” invoked by the Visualization System to get the sensors’ data.

It should be noted that after every single operation of the services notify the Monitoring System by a Message Broker (Kafka), notifying that a specific operation was conducted in the service. Even though this operation is not exposed in the Service APIs, this collaboration between our services and the monitoring service is also present.

This information is summarized in the following table.

Services	**Operations**	**Collaborations**
Artificial System	executeTask()	MAS Service (the invoker, by REST API) (the notifier, by Message Broker)
	getAllTask()	MAS System (the invoker, by REST API)
	getExperienceContent()	MAS System (the invoker, by REST API)
	notifyMonitoringSystem(action)	Monitoring System (the notified, by Message Broker)
Real System	updateData()	City System, getSensor() (to verify sensor exists, by REST API)
	getData()	Visualization System (the invoker, by REST API)
	notifyMonitoringSystem(action)	Monitoring System (the notified, by Message Broker)

Resilience Patterns

Our team implemented two resilience patterns in both of our services. The patterns implemented are the circuit breaker pattern and the timeout pattern. We can find the implementation of the timeout pattern in the RealSystemApplication.java file where we apply the pattern to the http client that is going to make the requests for our service. The circuit breaker can be found in the VerificationService.java file when our service makes a request to the Gateway.

- Circuit Breaker - this pattern enables our services to stop making requests for a certain time to the API Gateway if the gateway took too much time to respond multiple times during an interval of time. This way we can stop forcing the network and stop making redundant requests when we already know that the service is not going to answer.

- Timeout - this pattern enables our services to not wait too much time from responses of other servers that can be down or with too much throughput to handle.

Security Patterns

Regarding security patterns, we managed to implement two patterns in our services. Those patterns are authentication tokens and role-based authorization. We can find the implementation of the patterns tight together in our services in the class VerificationService. When the service receives a HTTP request it will run through a validation before it can execute the request where our security patterns are applied.

- Authentication Token - In the HTTP request besides the body of the request it also comes with a header where we can find the token that is going to be confirmed with the API Gateway, that serves as a trustful authentication source, to authenticate the client that is using our service. If the Gateway returns with a positive message (HTTP status ok, code 200), confirming the authentication of the client, the request will be executed otherwise it will be denied.

- Role-based Authorization - Besides the authentication token our team decided to implement the role-based authorization. When our service communicates with the Gateway to authenticate the client using the token, the response that it receives contains the http status code and also in the response body a Role. This role can then be used to apply a role-based authorization. An example : in the Artificial System only the clients that have the Researcher role can make requests.

Observability Patterns

As for the observability functionalities we implemented two patterns to enable a better comprehension of the state of our services. Those patterns are the health check endpoint and the distributed logging. The health check endpoint can be found in the controllers of both of our team services.

- Health Check Endpoint - This endpoint will be used by the monitoring team to ensure that all the services are up and running smoothly. It's a simple endpoint that returns the status code 200 to inform the monitoring team that the service is okay.

- Distributed Logging - this pattern is very useful to debug issues in a microservices application since all the services log their actions to a single module. We are using kafka to store and send our messages. Having a single module holding all the messages helps developers trace and track the origin of the problem.

Unit Test

Besides the postman tests required in this project our team implemented local unit tests.

The Real System service can also be tested locally with unit testing, so that the endpoints and the overall functionalities of the service can be assessed without needing the deployment of the outside services.

- In the 'application.properties' file in the resources:
  - the developer needs to disable/comment the PostgreSQL configurations (spring.datasource, and spring.jpa) because these configurations are set for the deployment in Heroku
  - the developer can replace the urls for 'City' and 'Gateway' services for 'localhost'. The project has controllers for these services that simulate their behavior.
  - the developer can disable logging to Kafka

The Artificial System service can be tested locally with unit testing, so that the endpoints and the overall functionalities of the service can be assessed without needing the deployment of the outside services.

- In the 'application.properties' file in the resources:
  - the developer needs to disable/comment the PostgreSQL configurations (spring.datasource, and spring.jpa) because these configurations are set for the deployment in Heroku

Open API Definition

Artificial System
https://artificial-system.herokuapp.com/swagger-ui/index.html#/

Real System
https://real-system.herokuapp.com/swagger-ui/index.html