

FLOOD WARNING

DISTRIBUTED SYSTEMS A.A. 2023/24

Michael Rubri - 0001146064 {michael.rubri@studio.unibo.it}
Alessandro Tullio - 0001052295 {alessandro.tullio@studio.unibo.it}

31 Gennaio 2024

Contents

1	Introduction	3
2	Requirements	4
2.1	User Requirements	4
2.2	Functional Requirements	6
2.3	Non-Functional Requirements	7
2.4	Implementation Requirements	7
3	Design	8
3.1	Domain	8
3.2	System Architecture	8
3.2.1	System Components	10
3.2.2	Emergency State Decision	13
3.3	Interaction of components	14
3.3.1	System initialization	14
3.3.2	Sensor removal	16
3.3.3	Adding sensor	17
3.3.4	Area redistribution	18
4	Implementation details	21
4.1	Technologies used	21
5	Validation	23
6	Deployment	25
7	Usages examples	27
8	Conclusions	31
8.1	Future works	31
8.2	What did we learned	32

List of Figures

2.1	Use Case Diagram	5
3.1	Diagram of the system components	9
3.2	Diagram of components by zone	10
3.3	Actors hierarchy	12
3.4	State diagram	14
3.5	Sequence diagram for system initialization	15
3.6	Sequence diagram for the initial insertion of an actor in the cluster structure	16
3.7	Sequence diagram for removing a sensor	17
3.8	Sequence diagram for adding a sensor	18
3.9	Sequence diagram for area redistribution	20
5.1	List of tests created	23
5.2	Tests passed successfully	24
7.1	Initial screen	27
7.2	Visualization of the zone of competence	28
7.3	Alarm management	28
7.4	Silenced alarms	29
7.5	Displaying non-competence zone	30
7.6	Visualization of different zone	30

Introduction

The project focuses on creating a distributed application with the aim of monitoring potential floods in a smart city. To achieve this goal, a predetermined number of "smart things" are installed in the city, which perceive certain environmental data. Specifically, various rain gauges are deployed to monitor the amount of rainfall, expressed in cubic millimeters, over a specific time interval. Each rain gauge is positioned in a specific location within a two-dimensional space. The city is divided into predefined zones, and each rain gauge operates in a single zone. The city's area is considered as a rectangle in Euclidean space. The zones must completely cover the city area; there cannot be any portion of space not managed by the system. The rain gauges transition from a dormant state to an alarm condition when the majority of sensors in a particular zone detect a value above a predefined threshold. Although the total number of rain gauges is fixed, each of them can fail, so the majority for a specific zone may change over time. When a zone is in an alarm state, the system must alert one of the nearby fire stations to the generated alarm. A single fire station can handle at most one alarm. For simplicity, each zone has only one fire station. The alarm generated by the sensors can be disabled only by a fire station. Therefore, even if precipitation falls below the allowed threshold, the alarm for a specific zone must remain active. Finally, firefighters have access to a GUI application through which they can view the statuses of various entities in the system and manage emergencies.

Requirements

The project's objective is to develop a distributed system that coordinates the presence of multiple entities dispersed over extensive areas. These entities collaborate to identify specific hazardous situations arising from heavy rainfall within a limited time frame and concentrated in a particular zone. The following outlines the system requirements formalized based on the identified categories of requirements.

2.1 User Requirements

The user requirements focus on the end user's experience and the manner of interaction with the application. Use case scenarios have enabled the definition of the primary actions that users can perform within the system. To effectively represent these scenarios and facilitate visual communication of what the system offers to the user, a use case diagram has been defined, as shown in Figure 2.1. This diagram has contributed to a clear and comprehensive visualization of user activities and interactions with the system.

The following are the listed user requirements.

1. The user must have access to a GUI with the following functionalities:
 - 1.1 It should display the current state of the user's designated area, where direct actions can be taken, as well as the states of all the zones into which the city has been divided.
 - 1.2 It should display the number of sensors in a specific zone.
 - 1.3 It should display the status of the fire stations, in particular whether they are currently occupied or available.
 - 1.4 It should allow a qualified operator to disable an alarm when appropriately managed.
 - 1.5 It should allow the user, regardless of the state of the zone, respectively to mute and unmute the interface.
2. The qualified operator must have the ability to add or remove new Sensor nodes in real-time without compromising the system's functionality. This flexibility allows for adapting the system to the changing needs of the smart city.

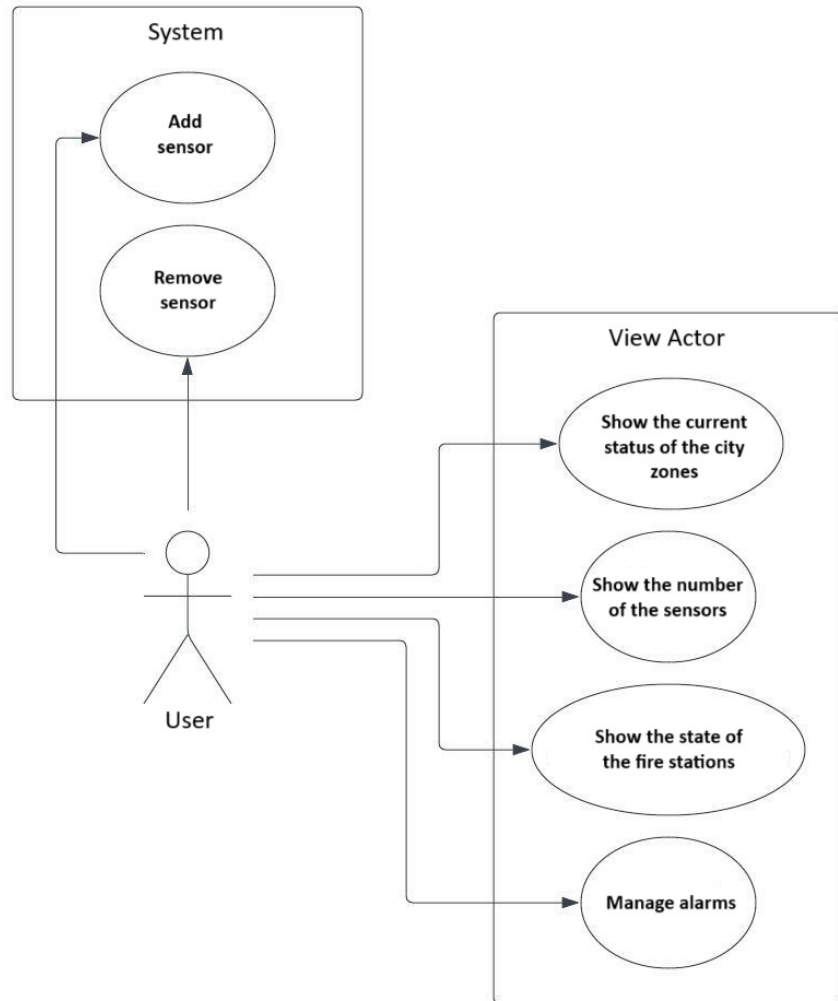


Figure 2.1: Use Case Diagram

2.2 Functional Requirements

Functional requirements outline the specific functions of the system, that is, what the system must do.

1. Deployer nodes:
 - 1.1 They create all the actors present in the system and are responsible for generating other entities both at the system's initialization and during its lifecycle.
2. Sensor nodes:
 - 2.1 They represent abstractions of the sensors.
 - 2.2 They collect information from the surrounding environment.
 - 2.3 They detect above normal levels of rainfall and send their status to the Coordinator Zone.
 - 2.4 They must have a distributed data structure that allows storing relevant information.
 - 2.5 They can be added or removed in real-time.
3. Barrack nodes:
 - 3.1 They periodically send their status to other Barrack nodes in the system.
 - 3.2 They periodically send the current state of the city and fire stations to View actors.
 - 3.3 They receive messages from the View actor at each interaction between the user and the graphical interface, updating the zone's status based on whether the emergency is currently being handled or not.
4. Coordinator Zone nodes:
 - 4.1 They receive information from Sensor nodes and determine if there is an emergency situation.
 - 4.2 They send their status to Barrack actors.
 - 4.3 They detect malfunctions in Sensor nodes and initiate the redistribution of the city area through a synchronized process to avoid inconsistency in data related to zone areas.
 - 4.4 They allow the addition or removal of Sensor nodes.
5. View nodes:
 - 5.1 They manage the interaction between an authorized user and the graphical interface.
 - 5.2 They send the response of the interaction to the Barrack actor.

6. The malfunction of a node should not compromise the functioning of the system.
7. It should be possible to introduce new modules without having to modify existing components.
8. Creation of the city:
 - 8.1 Ability to define the rows and columns to divide the area of the studied city.
 - 8.2 Ability to define the width and height of the city.
 - 8.3 Ability to define the initial number of sensors for each zone of the city.
 - 8.4 Ability to define the maximum accepted level of rainfall.

2.3 Non-Functional Requirements

The non-functional requirements define quality attributes, constraints, and general properties of the system.

1. Scalability: horizontal scalability to handle an increase in operational load.
2. Simplicity: the application must be easy to use, and the graphical interface must be intuitive.
3. Reliability: reliability and resilience are crucial aspects of the system. In the event of hardware failures (e.g., JVM failures) or network partitions, the system must be able to maintain the continuity of fundamental operations.

2.4 Implementation Requirements

Implementation requirements concern the technological and methodological aspects of the system's implementation:

1. Programming Language: the system must be implemented using the Java programming language.
2. Actor-Based Architecture: the implementation must follow the actor model paradigm using the Akka framework to manage interaction between nodes.
3. Code Documentation: the code must be adequately documented to enable clear understanding and facilitate future maintenance.

Design

3.1 Domain

The application domain revolves around actors cooperating within the distributed system. In particular, we have five types of actors, described in detail below:

- Deployer: father of the hierarchy, responsible for generating every other entity in the system.
- Sensor: models a generic rain gauge.
- Barrack: models a fire station.
- Coordinator Zone: models the node responsible for managing interactions between entities in a specific zone.
- View: models the graphical interface through which an authorized figure can manage emergencies.

3.2 System Architecture

The system architecture, shown in Figure 3.1, is peer-to-peer in nature and implemented through the use of an actor model, aiming to create a durable and robust system over time. The architecture is organized around interconnected components that enable monitoring and management of entities within the city. Each component plays specific roles within the system and interacts through well-defined interfaces. It is crucial to emphasize that for each zone into which the city is divided, a unique Coordinator Zone Actor, Barrack Actor, and View Actor are associated. These actors, along with the sensors belonging to the same city partition, form a cluster. It can be deduced that for each created zone, an independent cluster is formed, communicating and exchanging information with others.

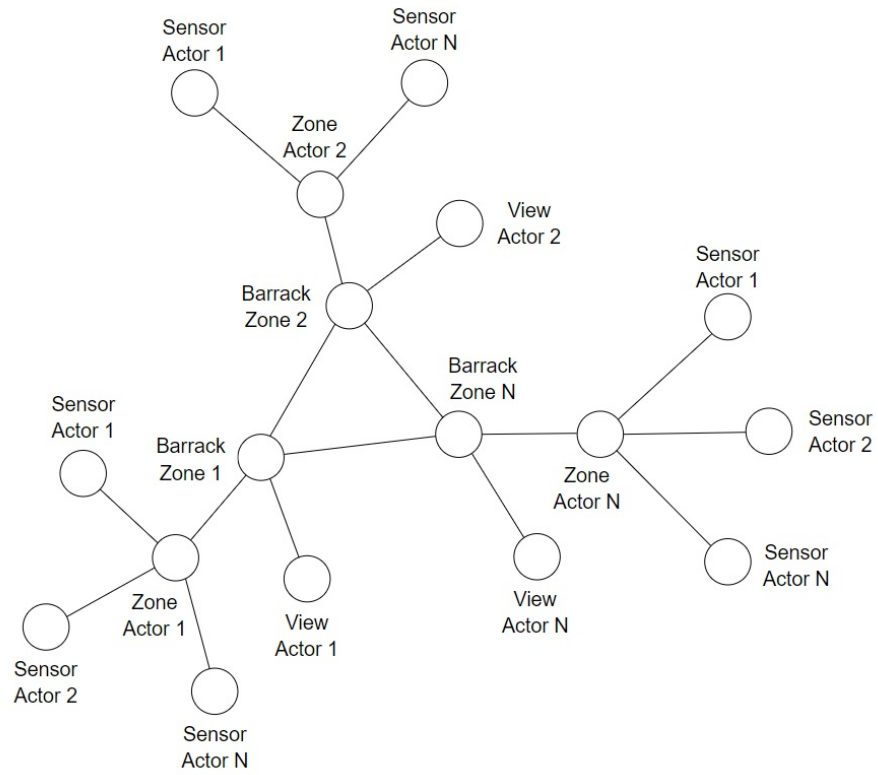


Figure 3.1: Diagram of the system components

In the Figure 3.2 below, the architecture of the running components and the network protocols used is presented. The system consists of a set of nodes, such as Coordinator Zone, Barrack, View, and Sensor, communicating with each other through the Akka protocol. Nodes communicating using this protocol are Actor Systems organized in a specific cluster that identifies a particular zone of the city, ensuring system scalability.

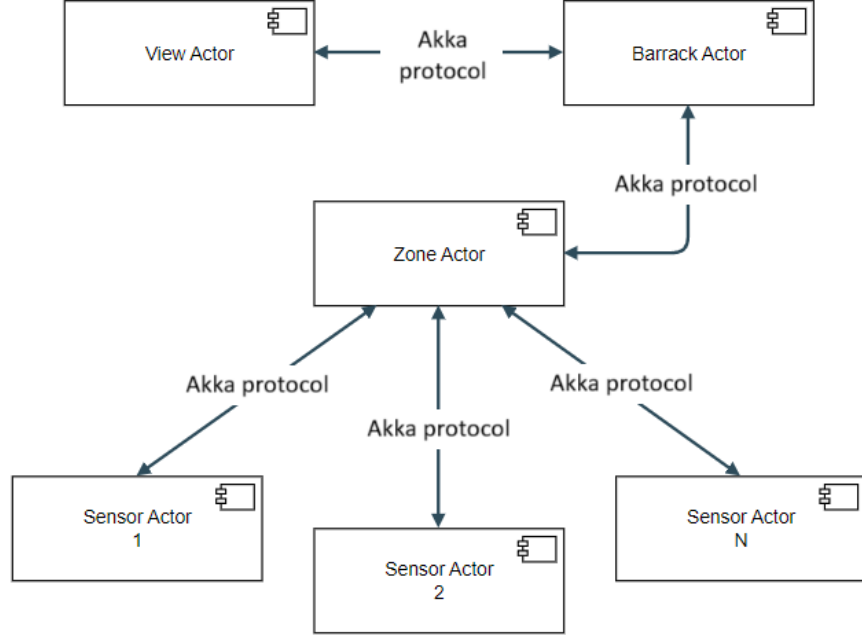


Figure 3.2: Diagram of components by zone

3.2.1 System Components

The central component of the system is the Coordinator Zone Actor, responsible for managing data processing and coordinating activities. Simultaneously, it serves as a communication point between Sensor and Barrack nodes. The Coordinator Zone acts as a mediator among various resources within the city, receiving data from sensors, processing them to determine if there is an emergency situation, and implementing zone redistribution if a sensor is intentionally removed from the system or due to hardware failures or persistent network partitions. This component also allows the insertion of a new sensor into the system during its normal operation. The Actor System related to the Coordinator Zone is of the *Singleton* type, this means that at any given moment, only one instance among all nodes specified with a particular role can actively participate in the cluster. This choice was determined by the desire to have a single point of responsibility for certain decisions, such as redistributing the area following the failure of a rain gauge.

In addition to the Coordinator Zone, the system consists of various actors, each playing a specific role in the process of acquiring, managing, and communicating

data related to resources.

The Sensor Actor is responsible for the periodic sampling of the water level and must continuously process such information regardless of the presence of additional sensors in the zone. It communicates with the Coordinator Zone Actor to send its own state. It can exclude itself from the cluster when it receives a stop message, depending on personnel needs, allowing for replacement or potential relocation within the zone.

The Barrack Actor interacts with both the Coordinator Actor and the View Actor. From the former interaction, it obtains the updated state of the specific zone of the city, which it uses along with its current state as a message to forward to the View Actor. From the latter, it receives a series of messages representing user actions interacting with the graphical interface, updating its state accordingly.

The View Actor manages interactions between the user and the graphical interface. Specifically, when an emergency occurs, depending on the decisions made by the personnel to handle the situation or not, these decisions are sent to the Barrack Actor.

The Deploy Actor is generated whenever it is necessary to add a generic node to the cluster. It is the parent of every actor present in the application and is named Guardian User, whose centralized purpose is to create the requested entity. In Figure 3.3, the hierarchy of actors has been defined.

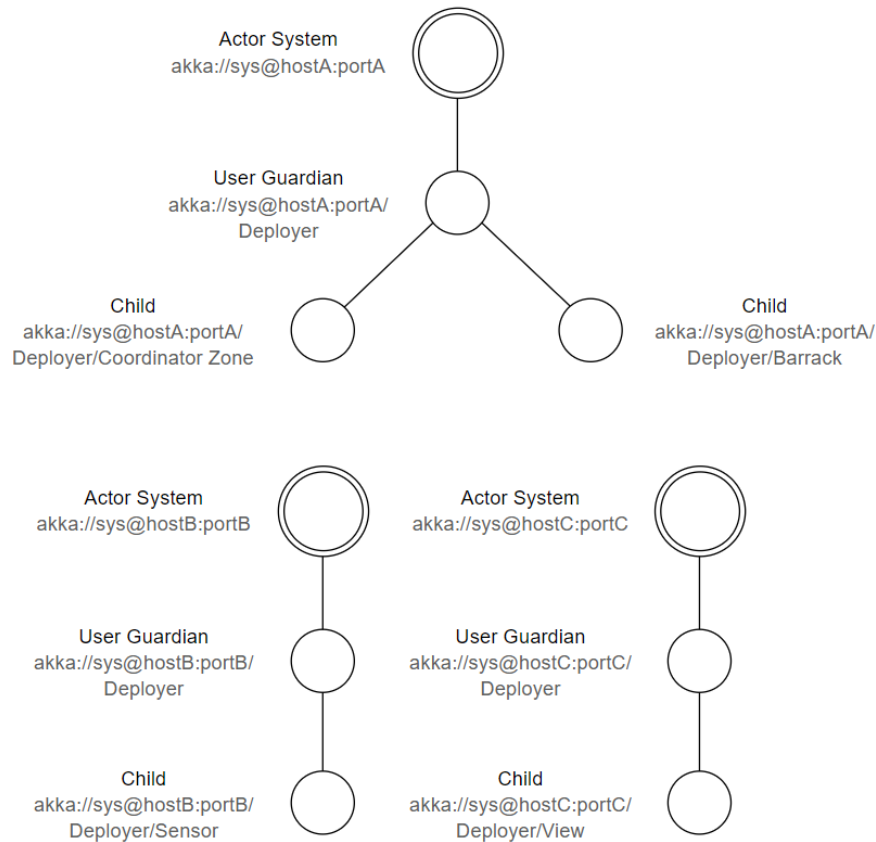


Figure 3.3: Actors hierarchy

The architecture involves clear and well-defined interactions between actors through the use of specific commands.

The Coordinator Zone Actor communicates with the Sensor Actor using the commands `RequestSensorDataMsg` and `StopSensor`, while communicating with the Barrack Actor through the `ZoneStatus` command.

The Sensor Actor communicates with the Coordinator Actor through the `SensorStatus` command.

The Barrack Actor communicates with the View Actor through the `CityStatus` command, while communicating with other Barrack Actors through the `BarrackStatus` command.

The View Actor communicates with the Barrack Actor through the commands `CommitBarrack`, `ClearBarrack`, `SilenceBarrack`, and `DesilenceBarrack`.

Every actor, except the Deploy Actor, also registers with a Receptionist service. This procedure is crucial when an actor needs to be discovered by another actor since it is not possible to insert a reference to it in a message. Each actor can register with one or more keys at the time of creation so that it can be discov-

erable by other nodes. This mechanism is based on the actor's message-passing model, and it is possible to search for and retrieve references of actors registered with a specific key. The registry is dynamic, meaning that new actors can be registered during the system's lifecycle. Moreover, all references of actors that are intentionally stopped or in the case the node they reside in is removed from the cluster are automatically removed, allowing for manual deletion from the registry. All messages sent and received by an actor on another node must be serialized.

3.2.2 Emergency State Decision

The various environmental sensors constitute the primary sources of data. Each sensor collects specific measurements and sends this information to the Coordinator Zone through a message. This communication enables the Coordinator Zone to process the data and provide a comprehensive view of environmental conditions.

This paragraph provides a detailed description of the conceptual design of the system, outlining the key components and their roles within the overall architecture to determine and manage the emergency situation. The operational cycle within the precipitation values monitoring system is as follows:

1. Sensors detect water levels at each predefined time interval.
2. The Coordinator Zone sends a message to the Receptionist at each predefined time interval to retrieve data from the Sensor nodes within the cluster.
3. Upon receiving the sensor references, it sends a message to each Sensor Actor to obtain the respective information.
4. The Coordinator Zone Actor evaluates the sensor measurements to determine whether to enter the alarm state. Specifically, it communicates to the Barrack Actor to enter an emergency state if the water level from the majority of sensors exceeds a critical threshold.
5. The Barrack Actor forwards the new state to the corresponding View Actor, allowing it to be displayed in the graphical interface, and to all fire stations in the city.
6. The graphical interface enables users to view the current state and allows them to take action by initiating emergency procedures and ending emergencies when the situation is managed.

The state diagram in Figure 3.4 illustrates the possible states that the Barrack Actor can assume.

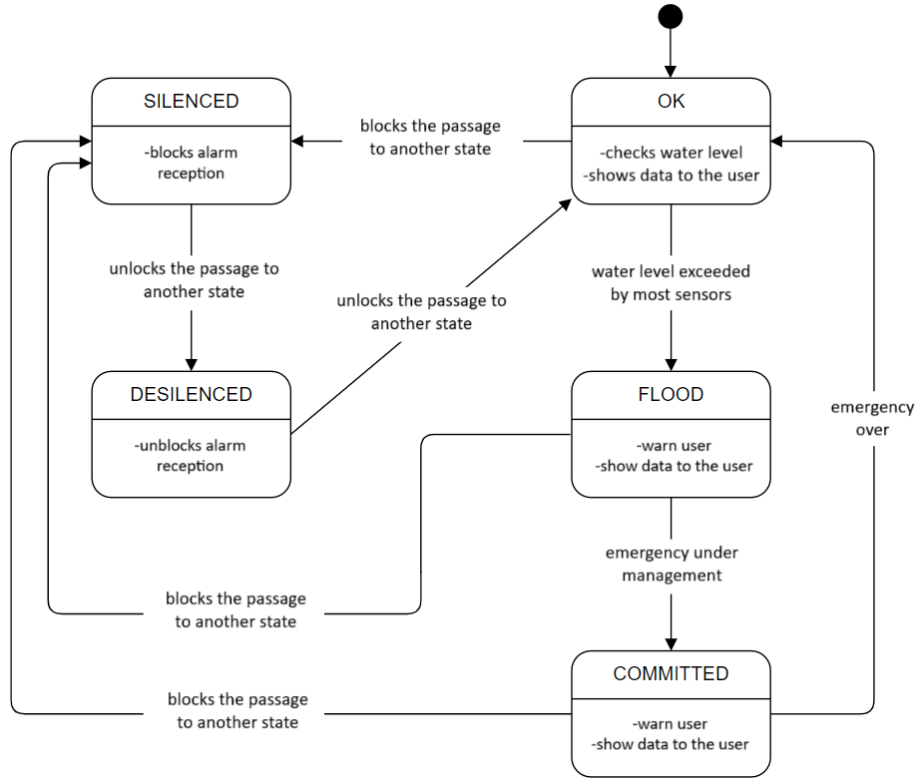


Figure 3.4: State diagram

3.3 Interaction of components

The overall behavior of the system is defined based on a series of interactions among the components. In particular, interactions can be grouped to define a specific aspect of this behavior.

3.3.1 System initialization

When the system is launched for the first time, it is necessary to initialize all data structures and create the cluster, as can be seen in Figure 3.5. The `SystemSimulation` class, through the `startup` method, communicates the port to use for initializing the cluster to the `Builder` class. At this point, the `Builder` creates all entities for each zone of the city. In particular, for each expected sensor (whose number is arbitrarily chosen in the initial phase but can vary during the system's lifecycle), an `Actor System` is created. This system, in turn, creates the `Deployer`, also known as the `User Guardian`, whose purpose is to create a child actor, the sensor in this case. Similarly, for each zone, a `View`

Actor, a Coordinator Zone Actor, and a Barrack Actor are created, each having a Deployer Actor as their parent. The seed nodes serve as the initial contact point to join the cluster. When a new node is created, it sends a message to all seed nodes and then sends a join command to the first node from which it receives a response. Once the process of joining the cluster for various nodes is completed, the seed nodes participate normally in the cluster in the same way as other nodes. In our case, the seed nodes are defined within the application.conf file.

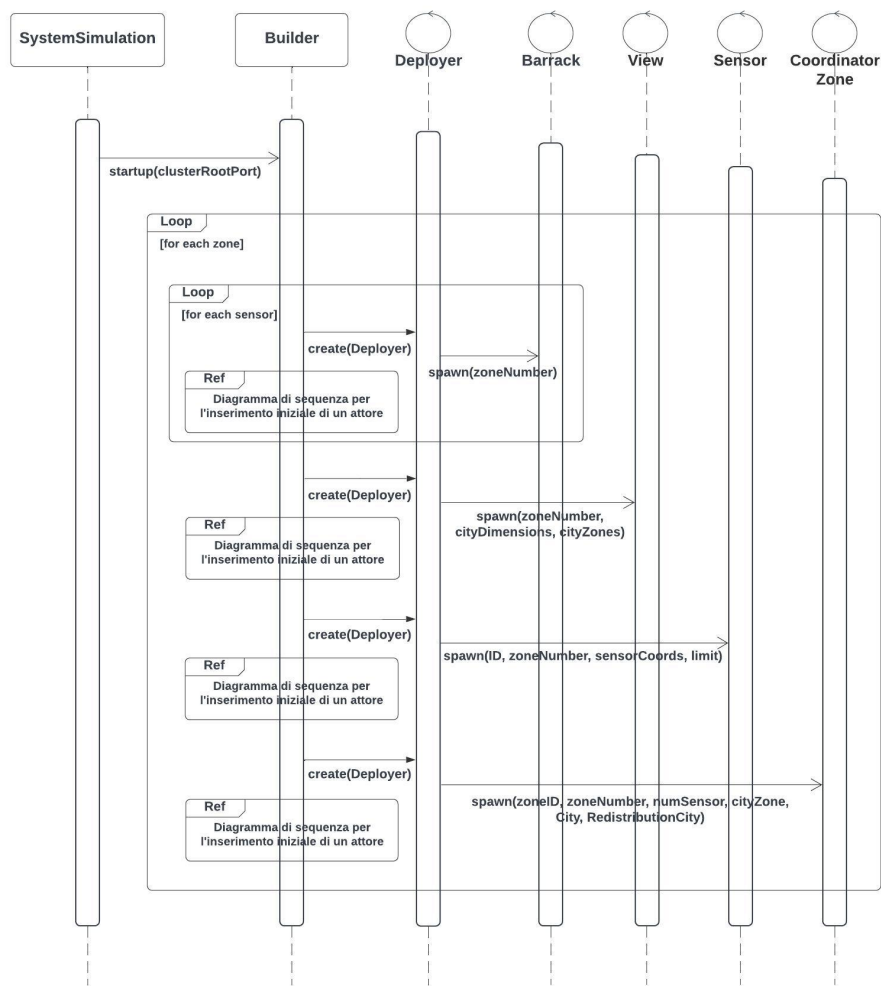


Figure 3.5: Sequence diagram for system initialization

At the system initialization, the Builder class, through the method exposed by the CalculatorZone class, defines each zone in which to divide the city of

interest, assigning each of them an identifier and their respective dimensions. Whenever an entity is created, this class adds the relevant information to the Cluster Structure (Figure 3.6). Regarding each sensor, information about its identifier and coordinates in space is also added. It is important to note that the information saved in the Cluster Structure pertains to the hostname and port of each node, allowing the establishment of a connection between cluster members and enabling the exchange of messages and information.

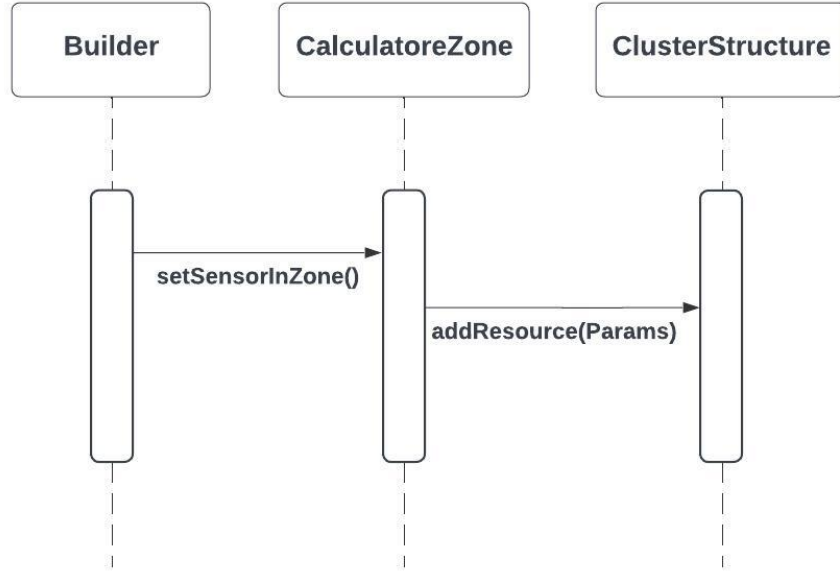


Figure 3.6: Sequence diagram for the initial insertion of an actor in the cluster structure

3.3.2 Sensor removal

During the system's lifecycle, it is possible to remove a sensor from the cluster of a specific zone (as shown in Figure 3.7), depending on the needs that may arise from the application users' requirements. It has been envisioned that if an individual responsible for sensor distribution deems the current location of the rain gauge unsuitable for detection, they have the option to remove it and place it in a new location, regardless of whether the zone remains the same or not, ensuring a high degree of freedom. Another motivation for introducing this feature is simply the ability to revise a sensor. To implement this procedure, the user must, from the command line, enter the keywords *stop sensor N*, where N indicates the number of a specific sensor among those available. In case the user does not input the phrase correctly or enters a non-existent number, the system continues its normal operation. Once the correct phrase is entered, the

Coordinator Actor sends an asynchronous message to the sensor, informing it to shut down. It then uses the method exposed by the Builder class to eliminate the sensor's information, such as IP address and port, from the cluster structure. Subsequently, using the method offered by the CityZone class, it removes the said rain gauge from the zone's list of sensors. This operation is executed through *Coordinated Shutdown* and ensures a *graceful exit*, informing the cluster members that the node is about to be removed. This functionality can be used appropriately only when the number of View Actors is equal to the number of generated interfaces.

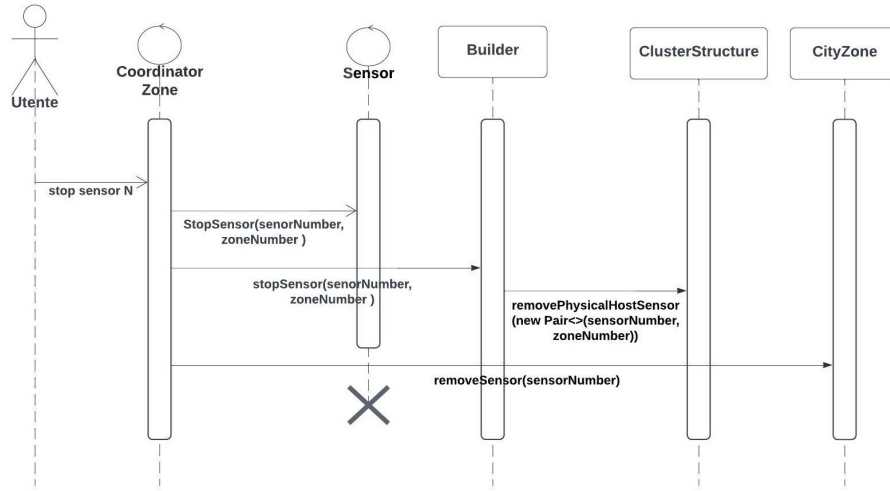


Figure 3.7: Sequence diagram for removing a sensor

3.3.3 Adding sensor

During the system's lifecycle, it is also possible to add a node to the cluster without affecting its operation. The introduction of this feature ensures a certain degree of scalability. The user must enter the keywords *add sensor* from the command line without additional information, as it will be the system's responsibility to determine the information to associate with the sensor. In case the user does not input the phrase correctly or provides additional unforeseen information, the system continues its normal operation. Once the phrase is entered correctly, the Coordinator Zone uses the method exposed by the Builder to initialize the process. The Builder class performs three sequential operations: it adds the new rain gauge to the list of sensors in the study area, calculating its identifier and new coordinates; it adds information, such as hostname and port number, to the cluster structure; it creates a new Actor System, specifying a Deployer Actor as the User Guardian. Once created, this actor generates the new sensor after receiving the specific message. This functionality can be used

appropriately only when the number of View Actors is equal to the number of generated interfaces.”

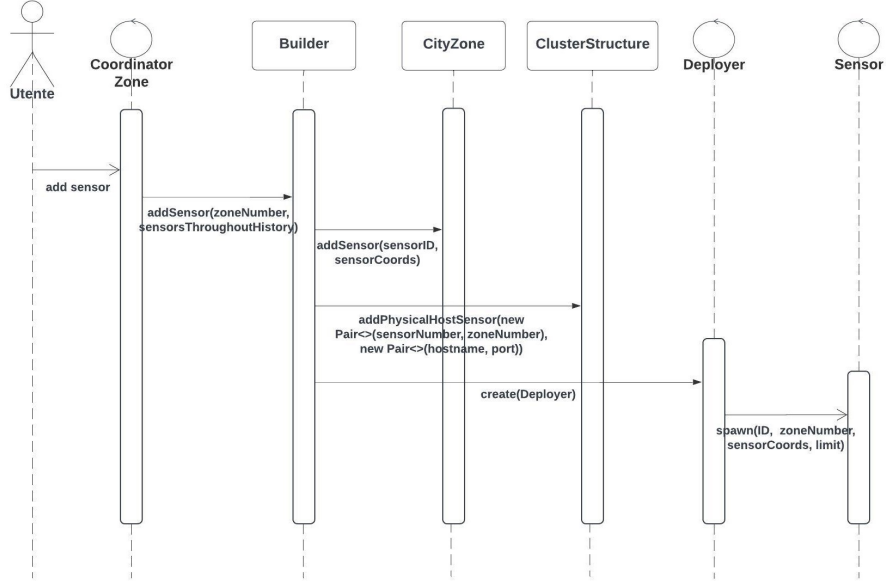


Figure 3.8: Sequence diagram for adding a sensor

3.3.4 Area redistribution

At system initialization, the number of sensors present in a specific zone of the city is established. However, the possibility has been considered that a sensor may fail or that persistent network partitions may prevent proper communication between nodes. A fundamental problem in distributed systems is the inability to distinguish between these two cases. A node may observe that there is an issue with another node, without being able to determine the cause, and moreover, it cannot know if the node will become available again. Another problem concerns the inability of a process to respond due to overload or the perpetual inability of a ready-to-execute process to obtain the necessary hardware and software resources, a condition known as *starvation*. Akka uses a failure detector to detect the onset of these problems based on the exchange of *heartbeats* between cluster nodes. A significant drawback of this procedure is the potential creation of multiple separate clusters, each of which is unable to see the others. To address this issue, *Akka Split Brain Resolver* has been employed. The implemented strategy in this system is the *keep majority*, which, in the event of two partitions, will remove the one with the fewer number of nodes. In case of a tie, the set of nodes with the lowest address is retained. Whenever a sensor is removed from the cluster, whether voluntarily or not, a

process of redistributing the area is initiated (as shown in Figure 5.1). The Coordinator Zone uses the method exposed by the 'RedistributionCityZone' class to calculate the new boundaries of the current zone and the neighboring zone. The neighboring zone is chosen based on the distance between the center of the sensors in the current zone and the edge of a neighbor, selecting the one with the greater distance. Once a neighboring zone is identified, the center of the sensors belonging to the latter is calculated, and the Euclidean distance between the two centroids is computed. This distance is divided by two, and consequently, the boundaries of both zones are updated, provided that the common side between them has the same length. Note that this mechanism will not be applied if it is not possible to determine the current zone or the neighboring zone, or if the latter has no sensors.

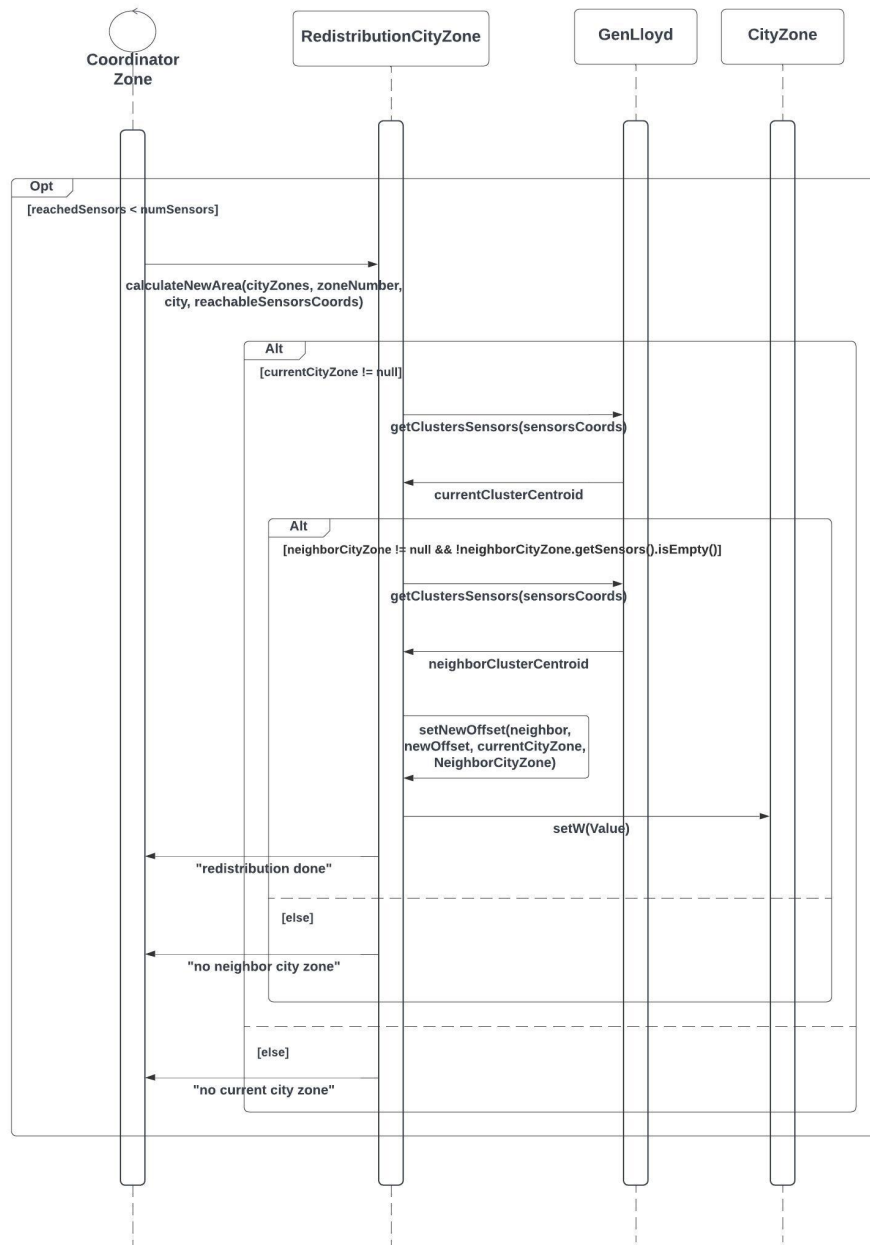


Figure 3.9: Sequence diagram for area redistribution

Implementation details

The system has been implemented as a multi-module project. The three main modules are:

- Distributed
- Javascript/Distributed-client
- Test

The Distributed module includes the code of the main project, while in the Javascript/Distributed-client module, is present the part related to the view. Finally, the test module was used to test the various components of the system

4.1 Technologies used

Below are listed the various technologies used within the project.

1. Java: high-level, object-oriented, statically-typed programming language that relies on its namesake runtime software platform, specifically designed to be as platform-independent as possible.
2. Gradle: is an open-source automation system for development, built on the ideas of Apache Ant and Apache Maven. It introduces a domain-specific language (DSL) based on Groovy, replacing the XML-based approach used by Apache Maven for project configuration. Gradle scripts can be executed directly, in contrast to Apache Maven's project definitions. Gradle utilizes a Directed Acyclic Graph (DAG) to determine the order in which processes can be executed.
3. Akka: is an open-source toolkit that provides a programming model for building distributed, concurrent, and reactive applications. Akka implements the actor model, where actors are fundamental units of processing in Akka and represent lightweight concurrent entities that communicate with each other through messages. This programming model simplifies the management of parallel and distributed processing efficiently.

4. Node.js: is an open-source, cross-platform runtime environment for executing JavaScript code, built on the V8 JavaScript engine from Google Chrome. Many of its core modules are written in JavaScript, and developers can write new modules in the same programming language. Originally, JavaScript was mainly used on the client side; Node.js enables the use of JavaScript to write server-side code as well.
5. Javascript: is a multi-paradigm programming language oriented towards events, used in both client-side web programming and server-side programming (Node.js) for creating REST APIs, desktop and embedded applications, websites, and web applications. Script functions, used in presentation logic, can be appropriately embedded in HTML files, JSP pages, or separate files with a .js extension, which are then called in the business logic.
6. Vue: is an open-source JavaScript framework in a Model-View-ViewModel configuration for creating user interfaces and single-page applications. Vue.js features an incrementally adoptable architecture that focuses on declarative rendering and component composition. Vue components extend basic HTML elements to encapsulate reusable code. In general, components are custom elements to which the Vue compiler associates functionality.

Validation

To test the developed modules, a combination of manual and automated tests using the JUnit framework in conjunction with Akka's TestKit was chosen.

JUnit is a testing framework for the Java programming language. It is used to write and run automated unit tests, which verify the behavior of individual units of code, usually methods or classes, to ensure they work as expected. JUnit uses annotations, such as `@Test`, `@Before`, `@After`, to define test methods and test execution configurations. The framework also provides a class called `Assertions` that offers a set of static methods to perform checks in tests.

To effectively test Akka applications, the "Akka TestKit" is used, which is a module specifically designed for testing actor interactions and simulating the concurrent execution environment.

Various tests that have been created are represented in the figure 5.1 below.

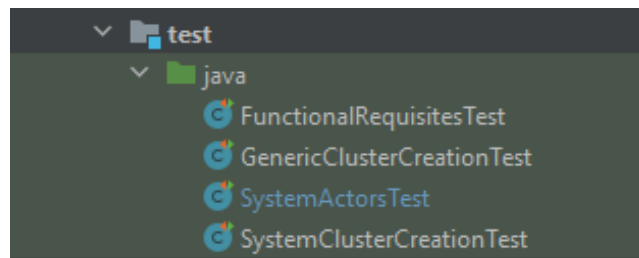


Figure 5.1: List of tests created

Below is possible see how all the tests pass correctly.

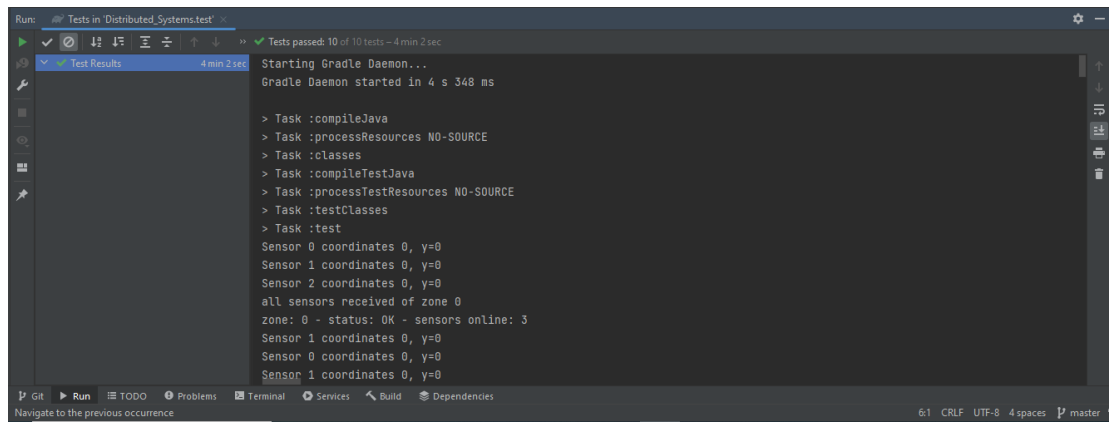


Figure 5.2: Tests passed successfully

Deployment

The project consists of a Java application and a graphical part in Vue. The Java application requires Java openjdk 17.0.9. To start the system, open a terminal and navigate to the project's package. Once inside the correct directory, you can follow two different paths. You can simply start the system using the command *gradle run* with default parameters. Additionally, the option to customize the parameters with which to start the application has been added, and these parameters include.

1. rows
2. columns
3. width
4. height
5. sensors
6. limit

These properties represent, respectively, the rows in which to divide the city area, similarly the columns in which to partition the city surface, the width of the city, its height, the initial number of sensors, and the maximum limit of the water level.

To define these properties, at the system startup, it will be necessary to use the command *gradle run -Pproperty1=value1 ... -PpropertyN=valueN* substituting one of the six values listed above for property and set value to the desired value. It is important to emphasize that in case the user does not enter a value correctly or does not specify all the properties, the default values will be used.

For the frontend implemented in Vue, the instructions to deploy the application are listed below:

1. You need to navigate to the 'javascript/distributed-client' package in the terminal
2. Run the command *npm install*. This command is used to install the project dependencies defined in the package.json file.

3. Run the command *npm run serve* to start the development server. Questo comando compilerà il progetto e avvierà un server di sviluppo locale. Once the compilation process is complete, a local URL should appear in the terminal.
4. To create an optimized version of the project for production, it's possible run the command *npm run build*. During this process, minimized, optimized, and ready-to-deploy files are generated for distribution on a web server.

Usages examples

Below are some possible examples of using our application. After starting a server instance, you can run and view the application.

The initial screen includes a map, and you can select your area of responsibility, as depicted in the screenshot below.

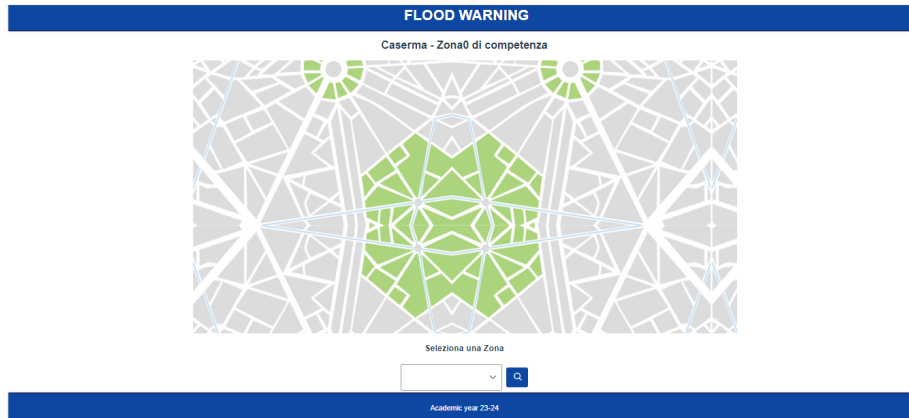


Figure 7.1: Initial screen

In the bottom part of the graphical interface, you can select your area of responsibility and access it by pressing the button identified by the magnifying glass.

In the screen that appears, you can view various values, such as the number of rain gauges, the current state of the area, the current state of the barracks, and whether there are currently partial data, meaning the number of sensors that have successfully communicated their state is less than the number of sensors belonging to the cluster. For each rain gauge, its ID is shown in the form 'zone number-sensor number,' the currently read water level value, and its corresponding limit value. When the majority of sensors read a value higher than the limit value, the area goes into an alarm state as shown in the figure.

Seleziona una Zona

Zona0

Q

🌙
SILENZIA ALLARMI

Status Zona0

⚠️ GESTISCI ALLARME

ALLARME

Status Caserma

LIBERA

Pluviometri

4

Dati Parziali

no

ID	VALORE	LIMITE
0-0	138.0	100.0
0-1	138.0	100.0
0-2	15.0	100.0
0-3	130.0	100.0

Academic year 23-24

Figure 7.2: Visualization of the zone of competence

When an emergency occurs, the user can decide whether to manage the emergency by clicking on the "GESTISCI ALLARME" button. During the alarm management, the status of the barracks changes from free to busy, while the area transitions to the "IN GESTIONE" state.

Seleziona una Zona

Zona0

Q

🌙
SILENZIA ALLARMI

Status Zona0

✓ ALLARME GESTITO

IN GESTIONE

Status Caserma

OCCUPATA

Pluviometri

4

Dati Parziali

no

ID	VALORE	LIMITE
0-0	119.0	100.0
0-1	86.0	100.0
0-2	1.0	100.0
0-3	61.0	100.0

Academic year 23-24

Figure 7.3: Alarm management

Following the decision to manage the emergency, a button labeled "ALLARME GESTITO" will appear. If pressed by an authorized user, it will allow the barracks to return to a normal state, indicating that the emergency situation has been successfully managed, thereby terminating it.

Additionally, the option to silence and unsilence alarms has been provided, as shown in the figure below. It is important to note that this mechanism is activated by clicking on the "SILENZIA ALLARMI" button and can be implemented independently of the state of the area or barracks. To resume normal operations, it will be necessary to click on the "DESILENZIA ALLARMI" button.

Seleziona una Zona

Zona0

▼

Q

🔊
DE SILENZIA ALLARMI

Status Zona0

OK

Status Caserma

LIBERA

Pluviometri

4

Dati Parziali

no

ID	VALORE	LIMITE
0-0	44.0	100.0
0-1	93.0	100.0
0-2	14.0	100.0
0-3	101.0	100.0

Academic year 23-24

Figure 7.4: Silenced alarms

While in their own competence area, users can directly influence the state of their barracks. However, they can also view the state of other areas without being able to make any decisions. To switch to the general view of a non-competence area, it will be necessary to select it from the available ones and click on the magnifying glass icon next to it.

Seleziona una Zona

Zona0

▼

Q

☾

Zona0

Status Zona0

Status Caserma

Pluviometri

Dati Parziali

OK

LIBERA

4

no

ID	VALORE	LIMITE
0-0	27.0	100.0
0-1	17.0	100.0
0-2	22.0	100.0
0-3	118.0	100.0

Academic year 23-24

Figure 7.5: Displaying non-competence zone

The interface will show the user the data related to a specific zone and the relevant barracks, allowing them to verify if the zone is in an emergency state and if it is being managed by the barracks.

Seleziona una Zona

Zona1

▼

Q

Zona1

Status Zona1

Status Caserma

Pluviometri

Dati Parziali

ALLARME

LIBERA

4

no

ID	VALORE	LIMITE
1-0	95.0	100.0
1-1	127.0	100.0
1-2	56.0	100.0
1-3	13.0	100.0

Academic year 23-24

Figure 7.6: Visualization of different zone

Conclusions

The result obtained is a consistent version of the application, which satisfies all the requirements set in the previous sections. The objective of creating a distributed system for monitoring possible flooding in a smart city has been achieved. The system was developed in a modular manner, so that it can be easily extended and maintained. Furthermore, the system was designed to be scalable and resilient, so that it can be used in a real context.

8.1 Future works

In terms of future developments, several features could be implemented, such as:

- Management of multiple monitored zones could be implemented. Multiple zones with different sensors could be added for monitoring.
- Addition of more sensors: Increasing the number of zones would inevitably involve adding more sensors.
- Different coverage of the zones: The division of zones could be different, with each zone divided following a different geometric shape.
- Integration of a real geographical map into the system. This would improve the modeling accuracy and allow for a more precise representation of the urban environment.
- Implementation of a proper database for data persistence. This way, each city could analyze its own risk level.
- Improvement of the graphical interface, as it was not particularly polished for this project since it was not among the main focuses of the course.

8.2 What did we learned

This project has allowed us to experiment with the development of a distributed system in all its aspects: from the design and planning phase to implementation and testing. Furthermore, it has provided us with an in-depth understanding of developing applications using sockets, revealing the reality behind such applications. We have learned to balance crucial aspects like security, scalability, and usability in a real-time communication project. During the creation of the distributed system, we acquired valuable skills in software development using the Akka framework. In particular, we delved into the actor model paradigm, which proved to be fundamental for designing and implementing highly scalable and resilient distributed systems. On the other hand, Akka provided a robust infrastructure for efficient actor implementation, enabling asynchronous communication through messages while ensuring fault handling and automatic supervision of the actors themselves. These skills allowed us to build a robust and highly performing distributed system, highlighting the value of the actor model paradigm in the realm of distributed development.