Forest Fire Detection using Multi-Agent Systems and Wireless Sensor Networks

Filippo Paganelli
filippo.paganelli3@studio.unibo.it

March 2020

In the urbanisation process, new technologies are playing a key role in improving the quality of life through sustainable resource management. These technologies are adopted to distribute and deploy in the environment the systems useful to carry out all the functionalities necessary for its monitoring and control. Wireless Sensor Networks are application-specific, and therefore must to involve both software and hardware. The aim of this project is the creation of a Multi-Agent System that allows the control of Wireless Sensor Networks by reducing as much as possible the dependence on specific scenarios. In order to test the results obtained, Forest Fire Detection was chosen as a case study. The whole work will be focused on the design of the Multi-Agent System and to the development of the software through the use of the framework Smart Python Agent Development Environment (SPADE). Anything related to the hardware part goes beyond the aspects related to the Distributed Systems and therefore the interactions of sensors and actuators with the environment will be simulated through data streams.

Contents

1 Goals/Requirements **3**
1.1 Scenarios 3
1.2 Self-assessment policy 4

2 Requirements Analysis **4**

3 SPADE **5**
3.1 Prosody IM 8

4 Design **9**
4.1 SensorStreamer 10
4.2 SensorAgent 11
4.3 DBConnectors 15
4.4 DBManagerAgent 16
4.5 TriggerAgent 18
4.6 StatisticsAgent 20
4.7 FrontEndAgent 22
4.7.1 Routes 24
4.7.2 Client Application 25
4.8 Interaction 26

5 Spreading of Information **30**

6 WSN Network Topologies **32**
6.1 Building the network topologies 33
6.2 Adopted Topology 34

7 Self-assessment / Validation **35**

8 Deployment Instructions **38**
8.1 GitLab Repository 38
8.2 MongoDB Community Server 38
8.3 ProsodyIM 39

9 Usage Examples **39**

10 Conclusions **41**
10.1 Future Works 42
10.2 Personal Opinion 42

1 Goals/Requirements

The choice of this project derives from the need to create a system with a real application by using tools I personally do not know in order to increase my knowledge as well as to learn the key concepts of Distributed Systems and Agent-Oriented Programming.
The objectives are the following:

- Providing reusability and generalisation to the system so that it can be used for scenarios other than the one chosen.
- Understanding and using patterns for good software design and implementation.
- Interacting with the environment through sensors and actuators.
- Indicating the presence, certain or possible, of a phenomenon (in this case a fire).
- Testing system operation with different network topologies and by applying parameter tuning.

1.1 Scenarios

The ability to interact remotely with the environment in which a system operates leads to an increase in productivity and a reduction in costs and dangerous situations. From a system like the one implemented in this project, the user is expected to perform various types of common actions that characterize the use of sensors and actuators:

- Reading values from sensors and activating the actuators by indicating when and how to operate.
- Displaying the history of the values read by the sensors but also where and when they were detected.
- Making statistics and making decisions with the historical data available.

In the chosen case study, the system provides the user with the data collected by the various WSN nodes, the related statistics and the actuators activation through a graphical interface, i.e. connecting via web to the system with a device that has a browser. This allows you to monitor forests in order not only to detect fires but also to understand the climatic conditions that caused it and that affected its growth. This system allows to increase safety by activating on-site alarms through the actuators and to reduce the timing of reversing on fires.

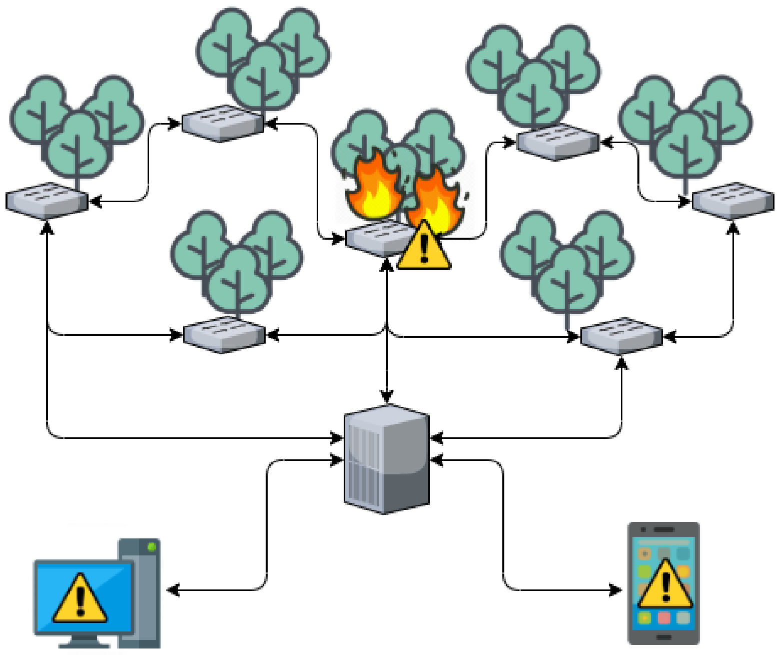


Figure 1: Example of the common application scenario of a WSN for Forest Fire Detection

1.2 Self-assessment policy

The effectiveness of the project results should be assessed on the basis of common criteria between the different scenarios, such as availability and consistency, but also on specific aspects such as compliance with deadlines in the case of hard real time systems.
In the chosen case study it will be necessary to consider the following aspects to evaluate effectiveness:

- Correctness of fire detection and related environmental data collected.
- Availability of data and the possibility of the user to view them through different devices remotely.
- Compliance with specific deadlines both for signaling fire detection and activating the alarm.

2 Requirements Analysis

The functional requirements of the system are the following:

- The sensors must be simulated and must continuously provide values in accordance with the chosen application scenario.
- The monitored phenomenon must be detected by the sensor readings and stored.

- Aggregations and transformations must be carried out on the stored data to create statistics on the functioning of the system.
- The user must be able to view data relating to measurements and phenomena reports.

The non-functional requirements of the system are the following:

- The multi-agent system part must be developed using the Smart Python Agent Development Environment (SPADE) framework.
- The system database must be developed using the NoSQL typology.
- The presentation of data to the user must be developed through a web application.

The non-functional requirement to use the SPADE framework introduces additional non-functional requirements in the project such as the use of XMPP for the management of communications between agents but also the use of certain tools for deployment and application testing such as Package Installer for Python (PIP), pytest and Virtual Environments (VENV).

The NoSQL database requirement is met using MongoDB technology which, due to its widespread use, provides wide support and availability of stable and well-functioning drivers. In the case of this project, the driver adopted is pymongo and its use will be deepened in the sections concerning MongoDB and the design and implementation of the database.

3 SPADE

Smart Python Agent Development Environment (SPADE) is a Multi-Agent System platform written in Python and based on instant messaging (XMPP) which it provides a great architecture to communicate in a structured way and solves typical problems of a platform such as user authentication (agents in this case). The choice of this framework derives both from the personal need to use the Python language for the first time and from the need to determine the similarities and affinities with the Java Agent Development Framework (JADE) framework, which is much more widespread than the SPADE framework.

The behaviors made available by the SPADE framework to define agents reflect those present in the JADE framework:

- OneShotBehaviour, designed to complete the task in one execution and useful for performing casual tasks.
- CyclicBehaviour, designed to never complete and useful for performing repetitive tasks.
- TimeoutBehaviour, designed to complete the task in one execution but activation is delayed by a specified period.

- PeriodicBehaviour, designed to perform tasks cyclically with a certain execution period.
- FSMBehaviour, allows more complex behaviours to be built.

The main difference with the JADE framework is the use of Extensible Messaging and Presence Protocol (XMPP) which is an open, XML-inspired protocol for real-time, extensible instant messaging (IM) and presence information. This technology offers by itself many features and facilities that ease the construction of MAS, such as an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML, just like FIPA-ACL. The generic architecture of a multiagent system based on the SPADE framework is very similar to that used in JADE because it too respects the model defined by FIPA.

The four main features required by the agent platform specification are:

- Agent Communication Channel (ACC), a mechanism which allows the agents and the platform itself to communicate with one another. This requirement is met through the Instant Messaging (IM) protocol found in XMPP.
- Agent Management System (AMS), a way for the agents to be registered in the platform and to be reachable for contact like White Pages service. This requirement is met through the Presence Information protocol found in XMPP.
- Directory Facilitator (DF), defined as a shared repository in which agents and services can discover services and where agents publish the services they offer, something like the Yellow Pages service. This requirement is met through the concept of template: when a behavior is added to an agent it is possible to associate a template with it. It is used by other agents in the system to send messages to specific behaviors that match the template.
- Agent Communication Language (ACL), a common language for all agents to communicate with. The SPADE framework supports FIPA metadata using XMPP Data Forms (XEP-0004). Once the message has been created, it is possible to define with the appropriate methods all the mandatory parameters defined by the FIPA ACL such as performative, sender, receiver, language, ”ontology”, ...

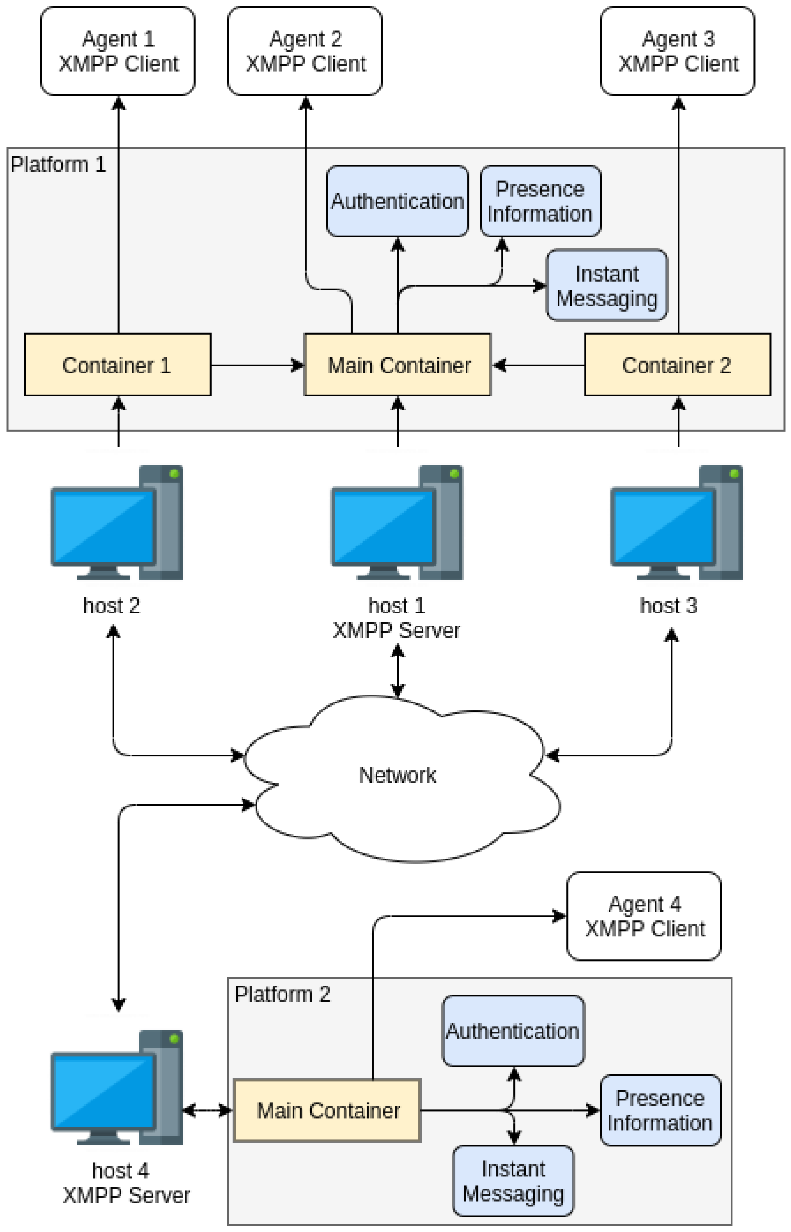


Figure 2: General architecture of a SPADE-based Multi-Agent System

By default, JADE always starts an HTTP-based Message Transport Protocol (MTP) with the initialization of a main container as a Agent Communication Channel (ACC). Internally, the platform uses a proprietary transport protocol called Internal Message Transport Protocol (IMTP) which is exclusively used for exchanging messages between agents living in different containers of the same platforms.

Each agent instance is identified globally by an Agent Identifier (AID) in the form:

```
<local-name>@<platform-name>
```

In SPADE communications take place, both between different platforms and between different containers in the same platform, according to the XMPP protocol: each agent is an authenticated XMPP client that is identified by an AID called Jabber ID (JID) in the form:

```
<username>@<server-domain>
```

Where username means the agent's local name and server-domain means the domain of the XMPP server where the agent is registered and authenticated, i.e. the platform.

3.1 Prosody IM

One of SPADE's features is the ability to use any XMPP server. On the official website [13] of the community that deals with developing and maintaining the XMPP standard there is a list of servers, most of which are open source, which can be used to run your XMPP services both on the Internet and in a local area network.

Tried some of the servers in the list, the one adopted in the project is Prosody IM [9], which is also the one recommended by the developer of the SPADE framework Javi Palanca.

4 Design

The design of the Multi-Agent System consists of a three-level virtual architecture. Virtual means that agents still belong to a container on a platform but can only communicate with agents of the same level or adjacent levels. We made this choice to better model the real structure of a Wireless Sensor Network where there are sensors and actuators that interact with the environment, servers that process the collected data and users who view the data through client applications.

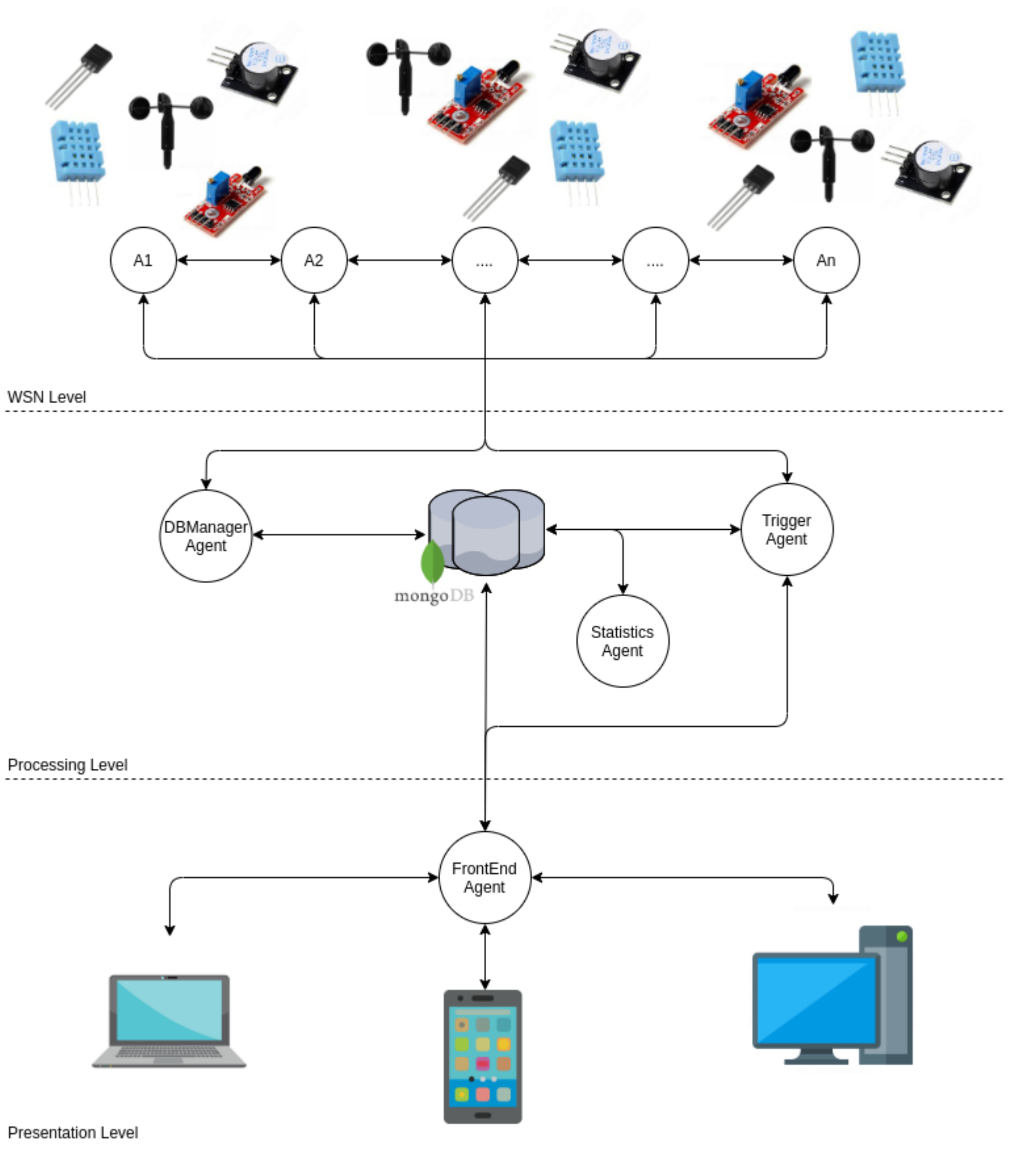


Figure 3: Virtual three-levels architecture of the Multi-Agent System

- WSN level, the agents of this level deal with the interaction with the environment obtaining the necessary values from the simulated sensors, carrying out actions on the actuators and spreading information among them. The communications with the upper level agents take place to store data and to receive commands on the activation of the actuators.
- Processing level, the agents carry out all the operations that concern the storage and manipulation of data. It is at this level that statistics on the trend of the collected values are calculated and decisions are made on the activation of the actuators.
- Presentation level, the main task of the agents of this level is to recover the data and present it in an appropriate manner to the user.

4.1 SensorStreamer

This section describes how we decided to design and implement sensor simulation in accordance with the requirements determined in the analysis phase.

For the detection of forest fires we decided to simulate a digital sensor for the detection of flames and analog sensors for the detection of other environmental data such as temperature, air humidity and wind speed. There are many other parameters to check such as soil moisture, wind direction and smoke but for the purpose of the project the ones indicated above were sufficient.

Parameter	Sensor Type	Sensor Example	Measure Unit	Data Type
flame	digital	KY-026	bit	boolean
temperature	analog	KY-013	Celsius	float
humidity	analog	KY-015	g/m^3	float
wind	analog	JL-FS2	Km/h	float

Table 1: Monitored parameters for detecting forest fires

Given the great availability of open source datasets on the network deriving from real applications [3], we decided not to randomly generate the final values, i.e. those read by the sensors, but to use files in Comma-Separated Values (CSV) format. By randomly generating the index of the row from which to take the value, it is instead possible to use the same file to obtain a different sequence of values each time.

For the creation of SensorStreamer entities, we decided to use the Abstract Factory creational design pattern. In this way it is possible to define new types of SensorStreamer and create them through the SensorStreamerFactory entity: the user, i.e. the SensorAgent, depends on the SensorStreamer interface and not on the specific implementation provided by the Factory.

The implementation of SensorStreamer made for the specific scenario of the project consists of FFDSensorStreamer. This entity allows to obtain the desired data using one or more DataStreamers. The latter is the basis of the simulation mechanism of the data stream: each DataStreamer corresponds to a CSV file and allows you to read a random value from the indicated column.

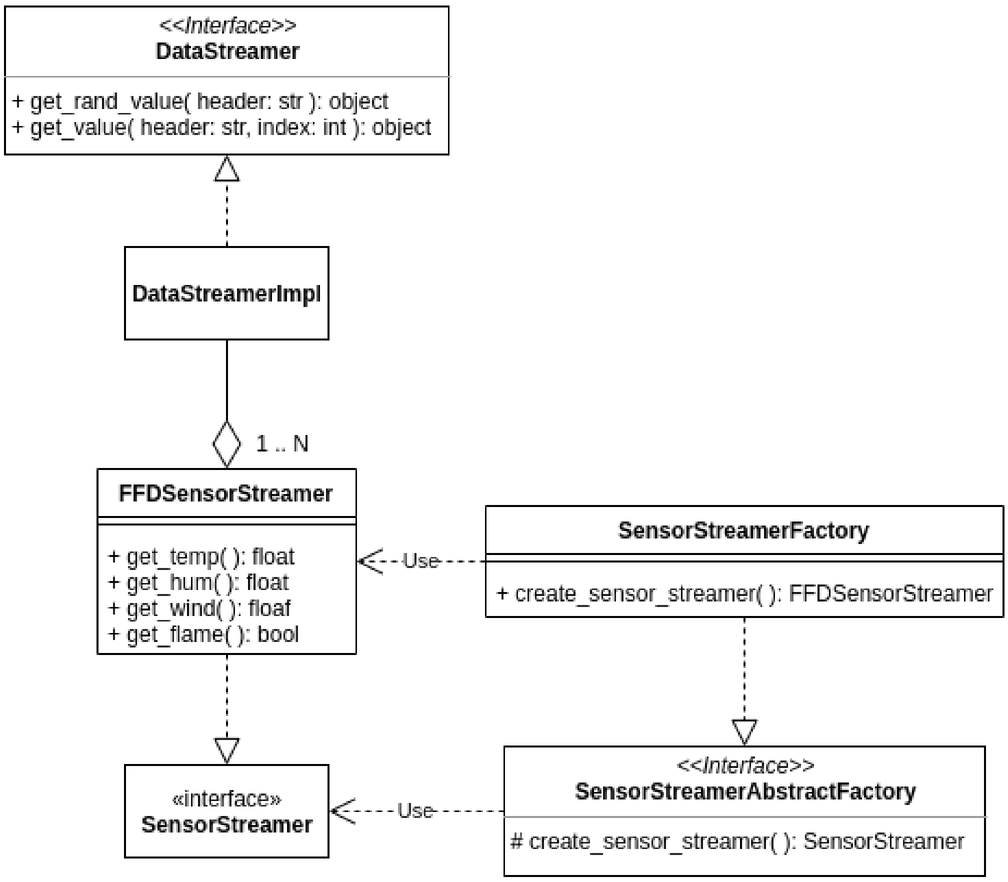


Figure 4: UML Class Diagram of SensorStreamer entity

4.2 SensorAgent

The SensorAgent agent, like all other agents, consists of a finite state machine obtained by extending the BaseFSM entity, which contains aspects common to all agents. These aspects mainly concern the definition of managers for subscription and unsubscription requests between agents.

The basic SensorAgent is defined by four states that represent the behavior of a sensor station in any application scenario. To reduce this model in a specific scenario, the behavioral design model of the strategy is used. In particular, the strategies that must be defined are:

- SensorStrategy, defines the logic with which the data is obtained but also which and how much data.
- ActuatorStrategy, defines the logic of the action to be performed on the actuators.

- SpreadingStrategy, defines the logic of information spreading, which includes for example the type of information and which agents to send it to.

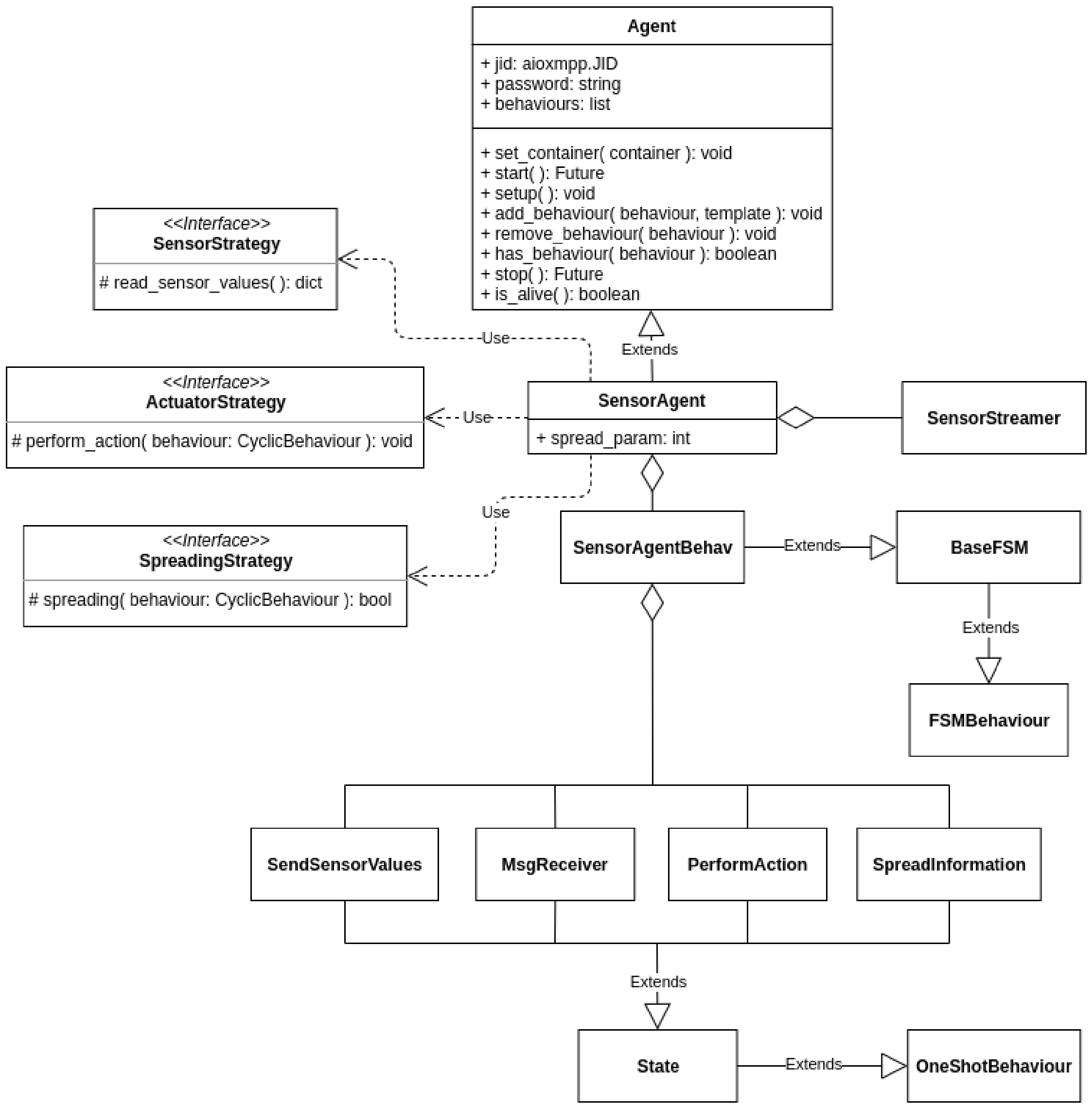


Figure 5: UML Class Diagram of SensorAgent entity

The specific SensorAgent created for the Forest Fire Detection, called FFDSensor-Agent, extends the basic SensorAgent by providing an implementation for each of the three strategies indicated above.

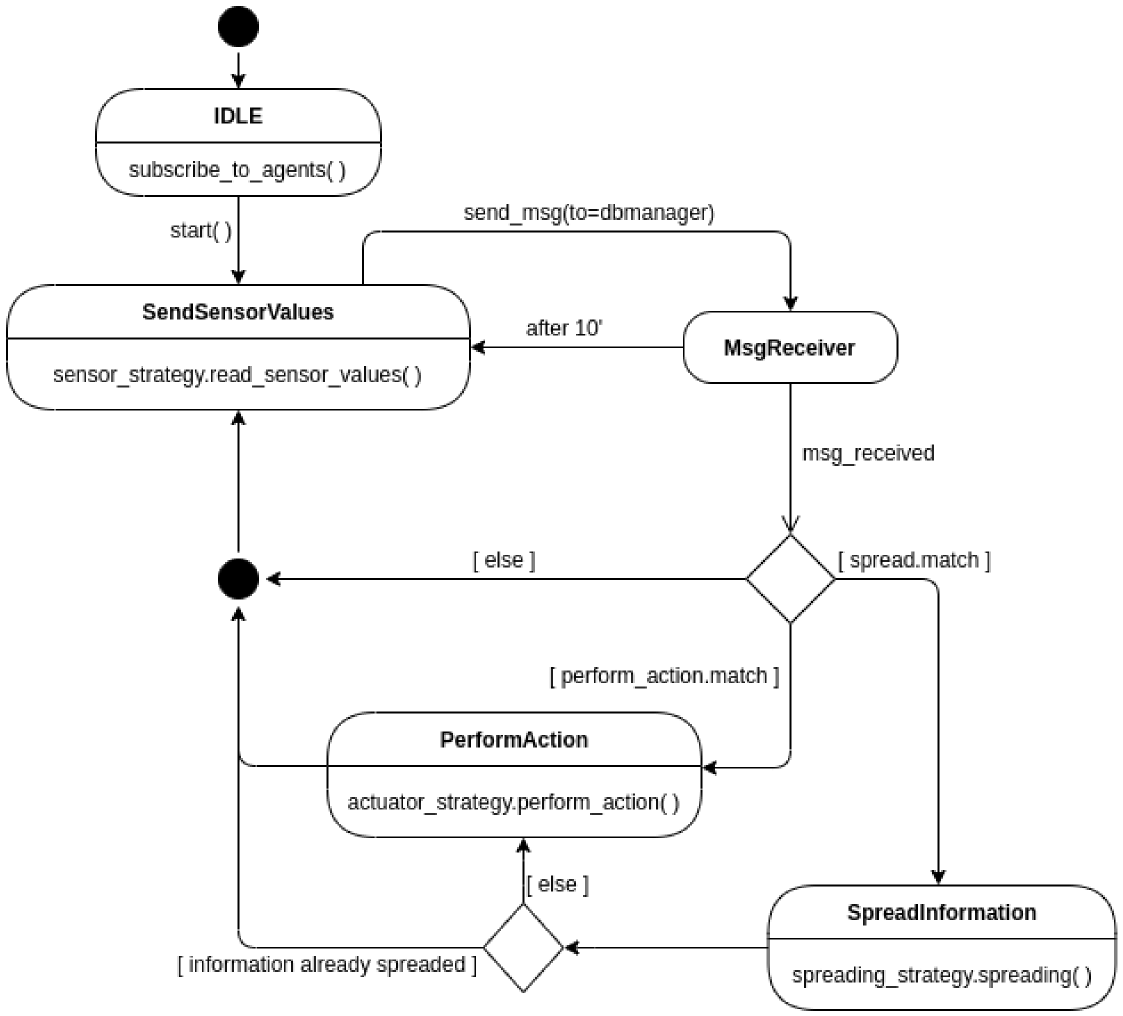


Figure 6: UML State Diagram of SensorAgent

The SensorStrategy uses the FFDSensorStreamer described above to obtain the values from the simulated sensors and returns a dictionary containing these values.

The ActuatorStrategy defines the action to be performed by adding a new behavior, called ActionBehav, to the agent. This behavior extends the PeriodicBehaviour construct: by specifying the period and duration it is possible to define an action that the agent will perform in parallel with respect to its normal execution flow.

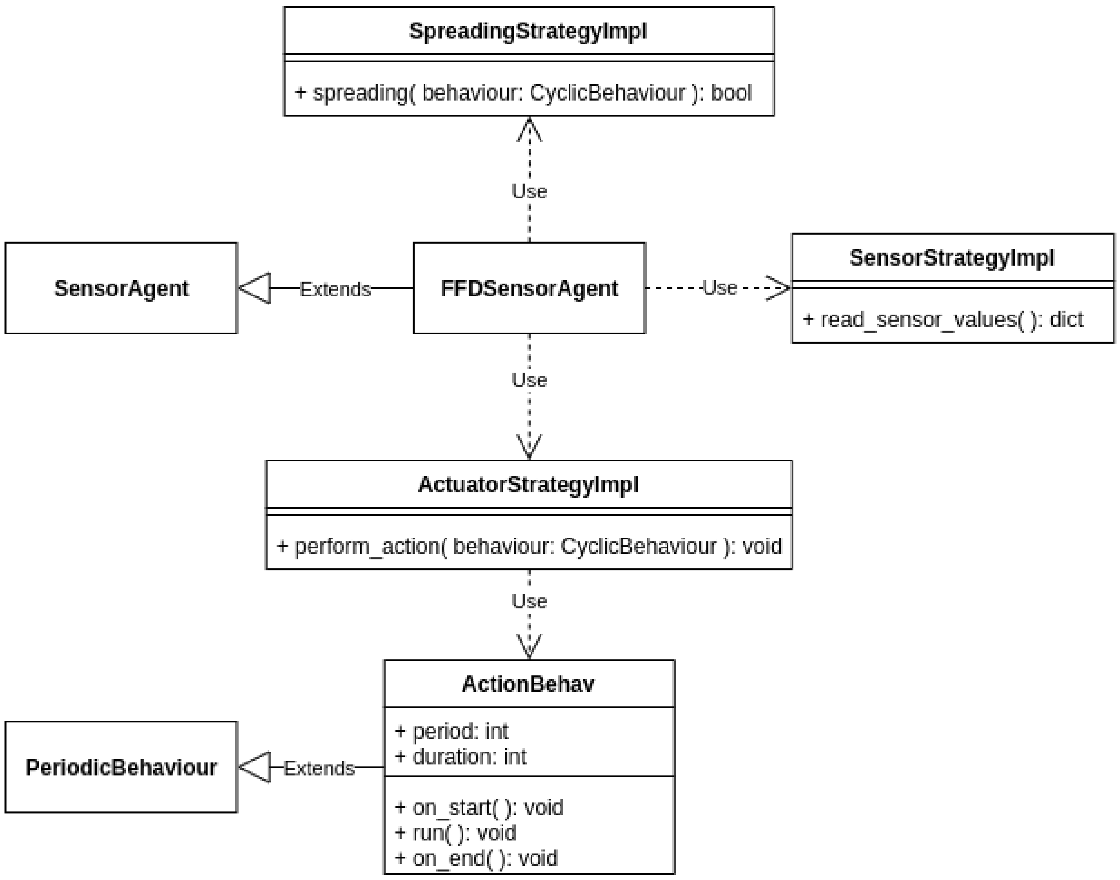


Figure 7: UML Class Diagram of FFDSensorAgent entity

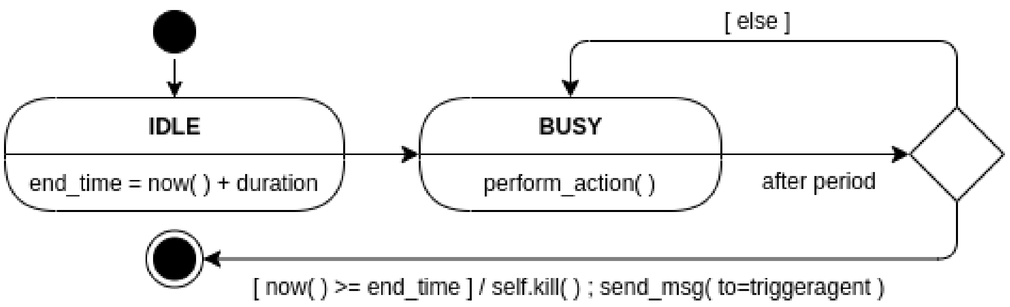
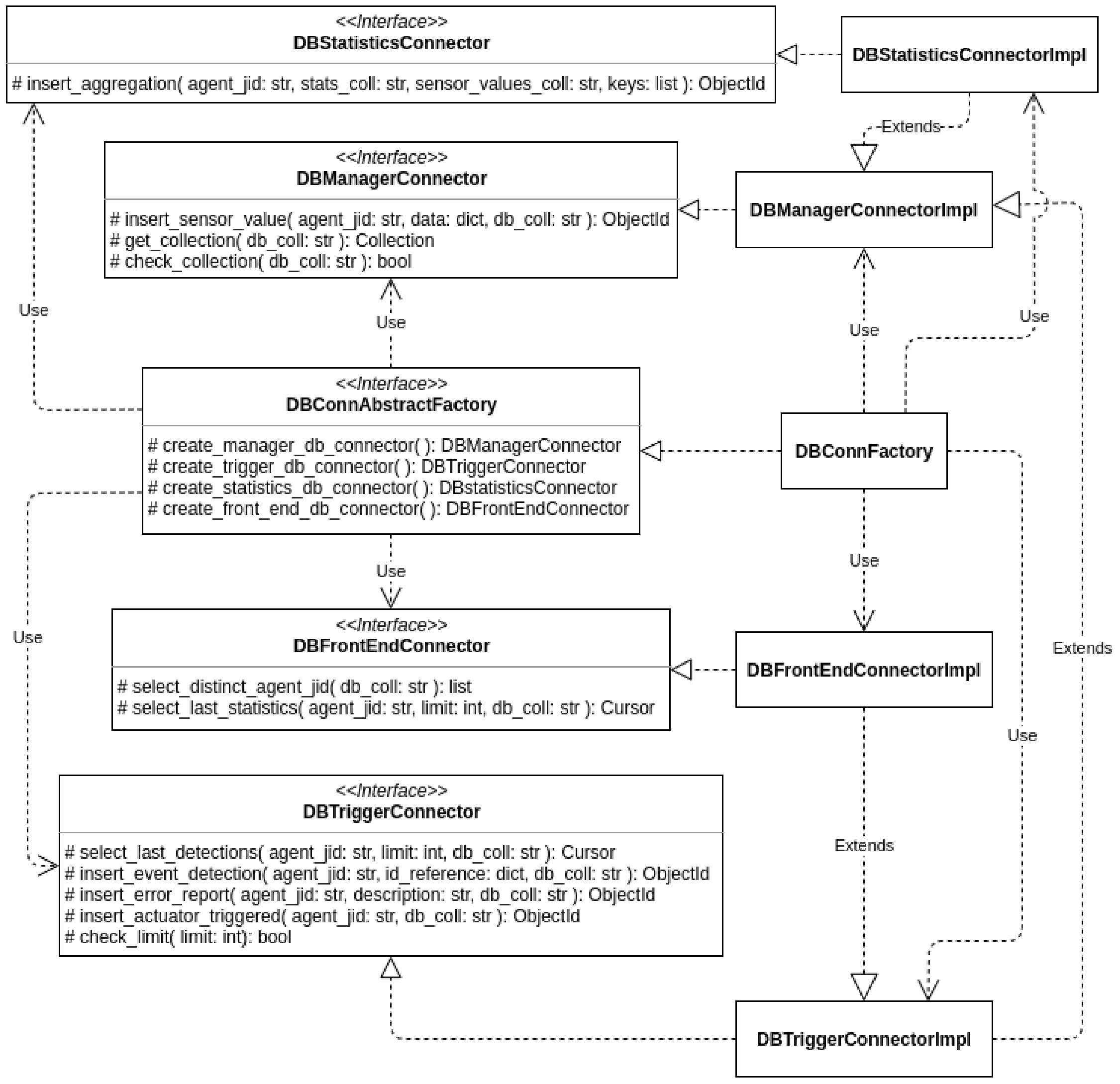

[else]
IDLE
end_time = now( ) + duration
BUSY
perform_action( )
after period
[now( ) >= end_time] / self.kill( ) ; send_msg(to=triggeragent)

Figure 8: UML State Diagram of ActionBehav entity

4.3 DBConnectors

Any other agent in the system other than the SensorAgent requires access to the database to operate and therefore a specific connector that provides the suitable constructs. The DBConnAbstractFactory entity, that also uses the Abstract Factory creational design pattern, provides the constructs to create the connector suitable for the agent and allows to provide a different implementation according to the needs of the application scenario.



<<Interface>>
DBStatisticsConnector
# insert_aggregation(agent_jid: str, stats_coll: str, sensor_values_coll: str, keys: list): ObjectId
DBStatisticsConnectorImpl
Extends
<<Interface>>
DBManagerConnector
# insert_sensor_value(agent_jid: str, data: dict, db_coll: str): ObjectId
# get_collection(db_coll: str): Collection
# check_collection(db_coll: str): bool
DBManagerConnectorImpl
Use
Use
Use
Use
<<Interface>>
DBConnAbstractFactory
# create_manager_db_connector( ): DBManagerConnector
# create_trigger_db_connector( ): DBTriggerConnector
# create_statistics_db_connector( ): DBstatisticsConnector
# create_front_end_db_connector( ): DBFrontEndConnector
DBConnFactory
Use
Use
Use
<<Interface>>
DBFrontEndConnector
# select_distinct_agent_jid(db_coll: str): list
# select_last_statistics(agent_jid: str, limit: int, db_coll: str): Cursor
DBFrontEndConnectorImpl
Extends
Use
<<Interface>>
DBTriggerConnector
# select_last_detections(agent_jid: str, limit: int, db_coll: str): Cursor
# insert_event_detection(agent_jid: str, id_reference: dict, db_coll: str): ObjectId
# insert_error_report(agent_jid: str, description: str, db_coll: str): ObjectId
# insert_actuator_triggered(agent_jid: str, db_coll: str): ObjectId
# check_limit(limit: int): bool
Extends
DBTriggerConnectorImpl

Figure 9: UML Class Diagram of DBConnectors

4.4 DBManagerAgent

The DBManagerAgent entity represents a central point within the distributed system: it is in fact the only entity that receives all the values from the SensorAgents and inserts them in the database after processing. This manipulation of data received prior to storage is defined using the Strategy behavioral design pattern. This is the point where to insert the various checks according to the values that are measured and the application scenario.

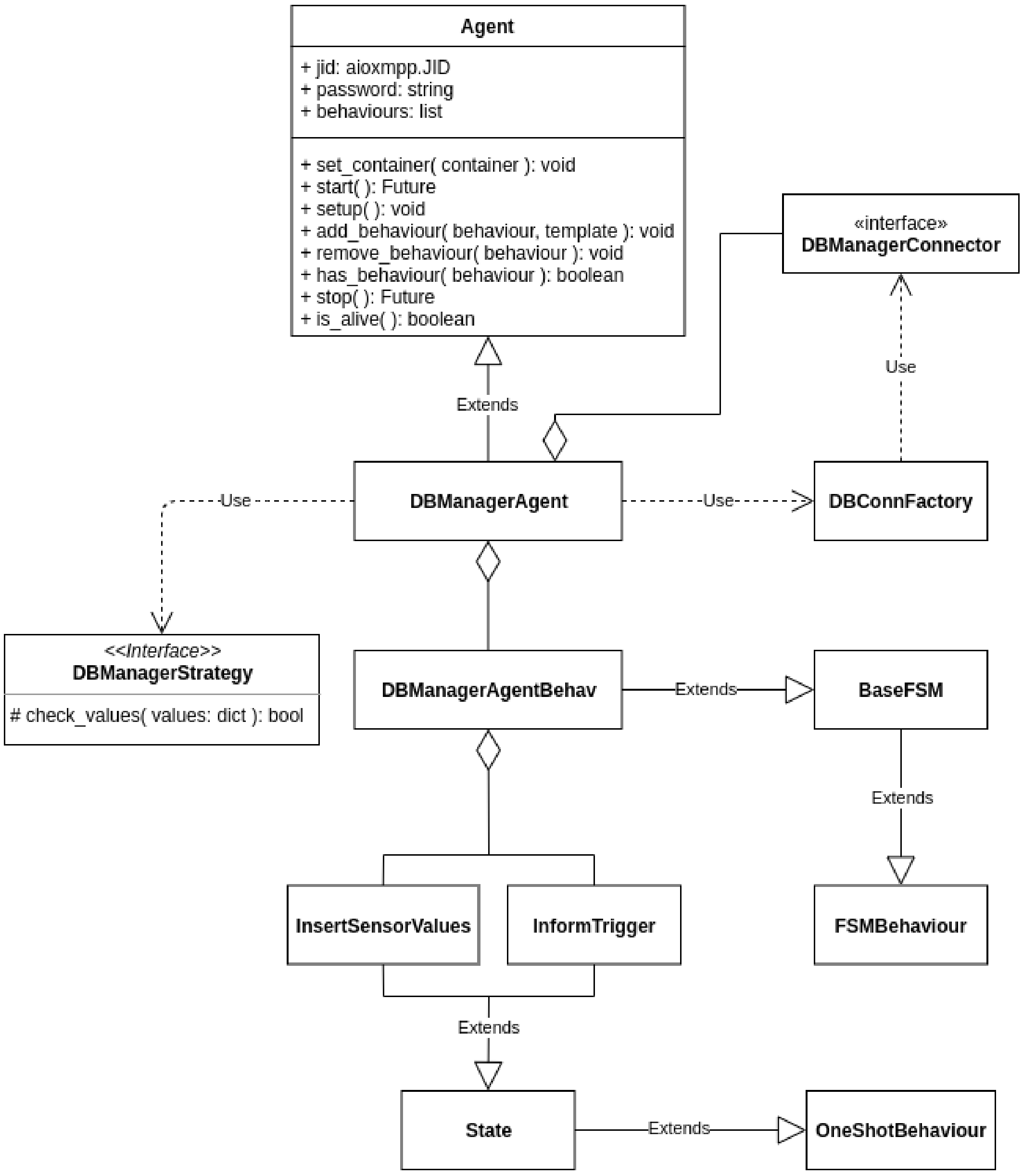


Figure 10: UML Class Diagram of DBManagerAgent entity

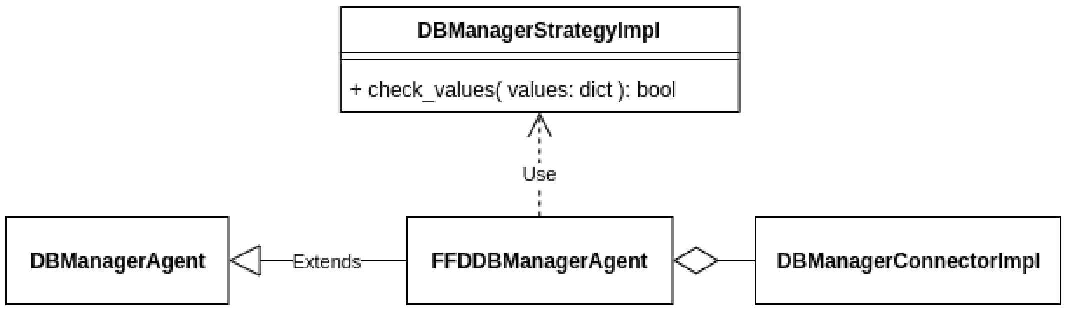


Figure 11: UML Class Diagram of FFDDBManagerAgent entity

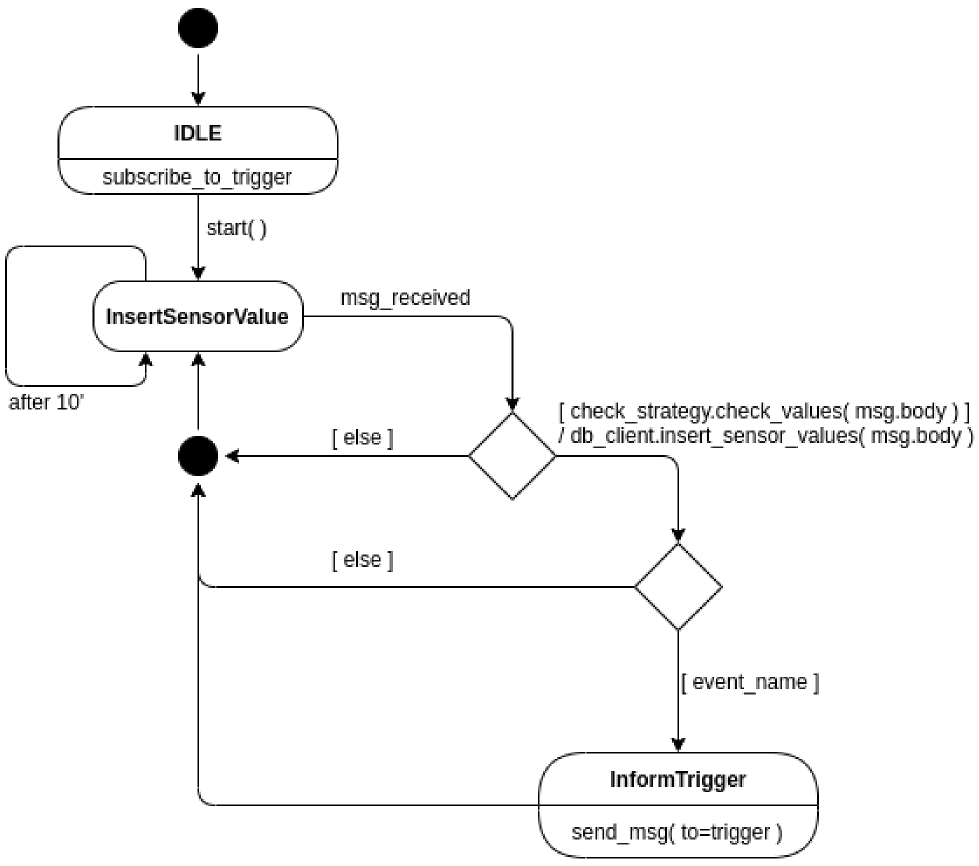


Figure 12: UML State Diagram of FFDDBManagerAgent entity

4.5 TriggerAgent

The TriggerAgent entity is responsible for checking the detections made by the SensorAgents and indicating, in the event of a positive detection, the actuation of the actuator.

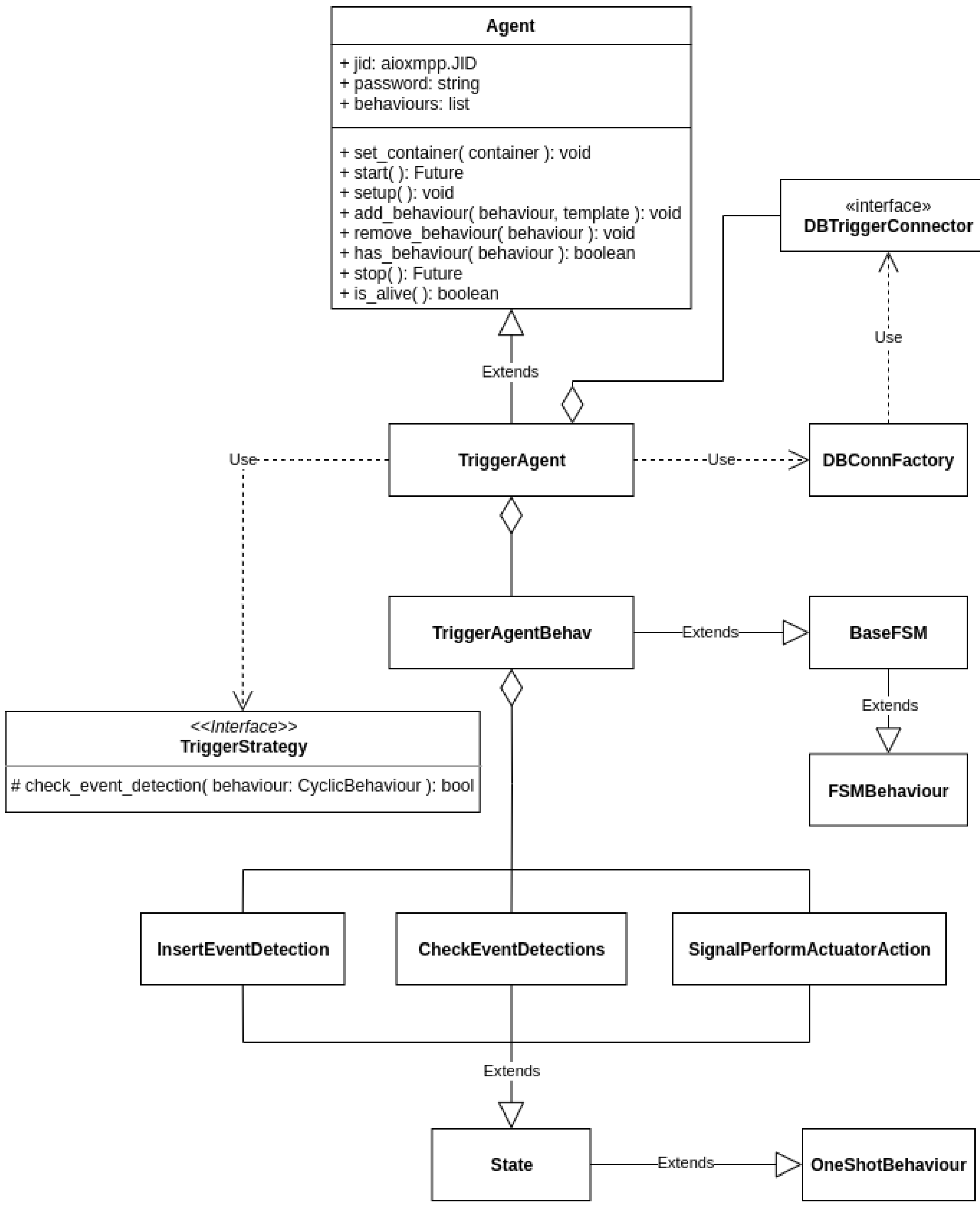


Figure 13: UML Class Diagram of TriggerAgent entity

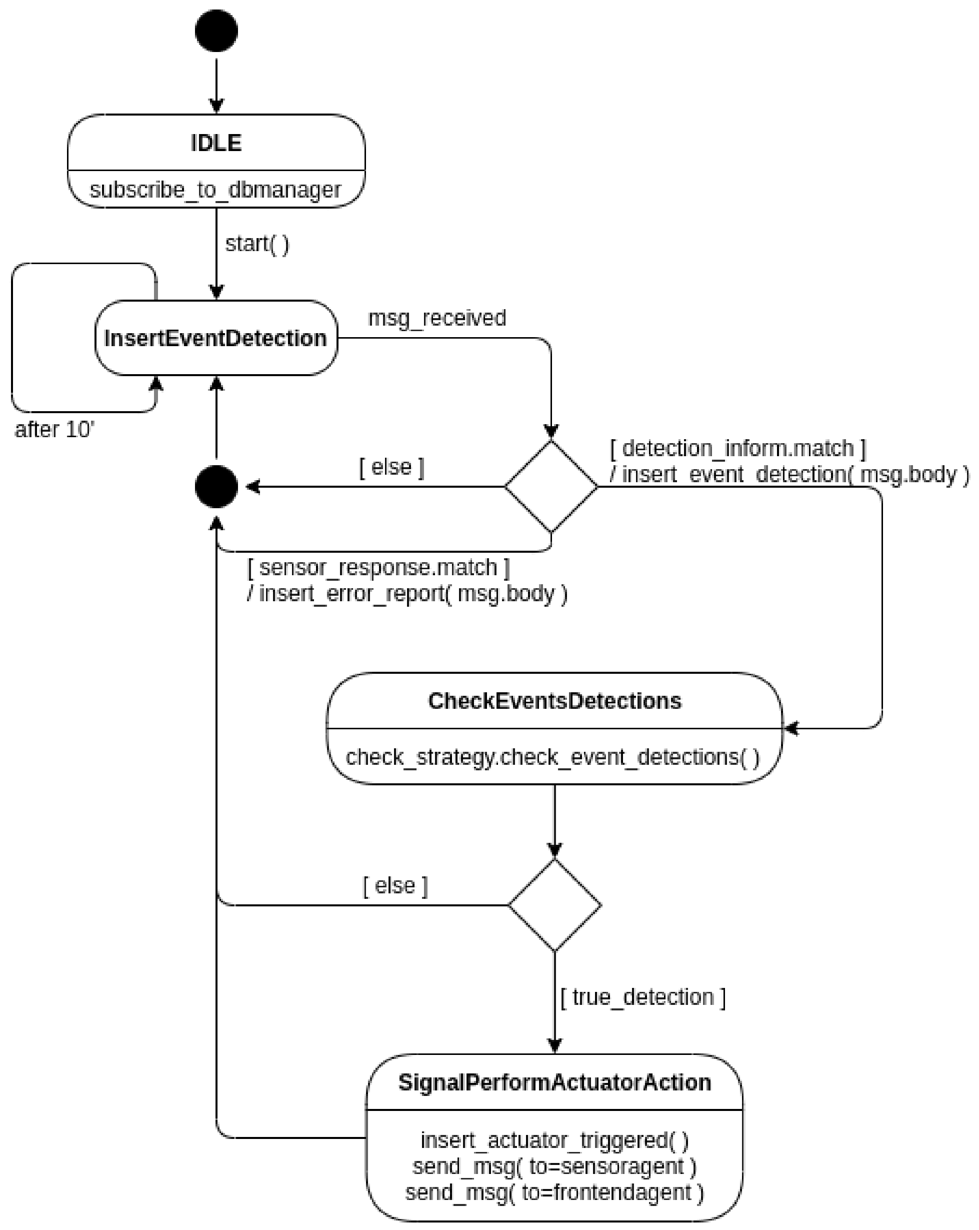

IDLE
subscribe_to_dbmanager
start( )
InsertEventDetection
msg_received
after 10'
[else]
[detection_inform.match]
/ insert_event_detection(msg.body)
[sensor_response.match]
/ insert_error_report(msg.body)
CheckEventsDetections
check_strategy.check_event_detections( )
[else]
[true_detection]
SignalPerformActuatorAction
insert_actuator_triggered( )
send_msg(to=sensoragent)
send_msg(to=frontendagent)

Figure 14: UML State Diagram of TriggerAgent entity

This agent avoids polling the database to check if there are new detections as it receives a notification from the DBManagerAgent agent but still requires access to the database to insert the detections considered valid and therefore also the actuation of the actuators. The connection to the database takes place through the entity DBTriggerConnectorImpl created through the already described AbstractFactory.

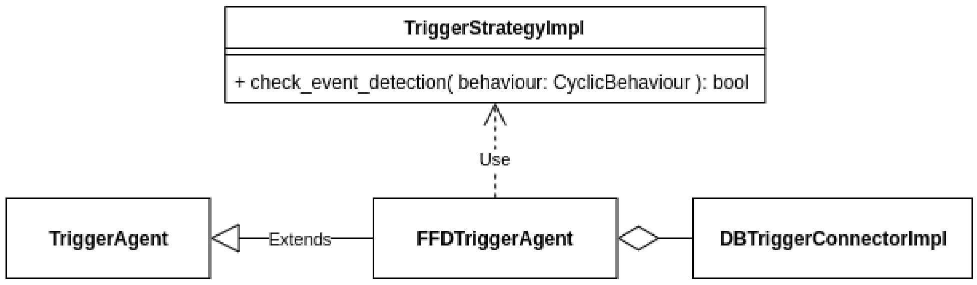


Figure 15: UML Class Diagram of FFDTriggerAgent entity

This agent implements the logic to determine whether the detections made by the sensors are true or spurious. The implementation provided by the TriggerStrategyImpl instance makes use of two parameters in order to verify the detections when receiving the message from the DBManagerAgent:

- NUM_EVENTS_DETECTIONS, represents the number of most recent detections to consider for a given SensorAgent.
- DELTA_TIME, maximum time lapse (in seconds) between two consecutive detections.

A detection is considered true and not spurious if the time difference between each pair of consecutive detections, between the NUM_EVENTS_DETECTIONS considered, is less than or equal to DELTA_TIME: in this case, an appropriate message will be sent to the SensorAgent that carried out the detection and a new document containing the related information will be stored in the database.

4.6 StatisticsAgent

The StatisticsAgent agent deals with aggregations on the data stored in the database in order to obtain useful data to know the progress of certain parameters.

Given the large amount of data that must be manipulated at each aggregation, this agent extends the classic BaseFSM entity by performing periodic behavior in order to set the appropriate execution period to perform all operations correctly.

Again the agent uses a special connector created using the DBConnFactory previously described.

The logic with which the statistics are created is not defined using a behavioral design pattern because it is possible to define it directly in the implementation of the specific database connector created through DBConnFactory.

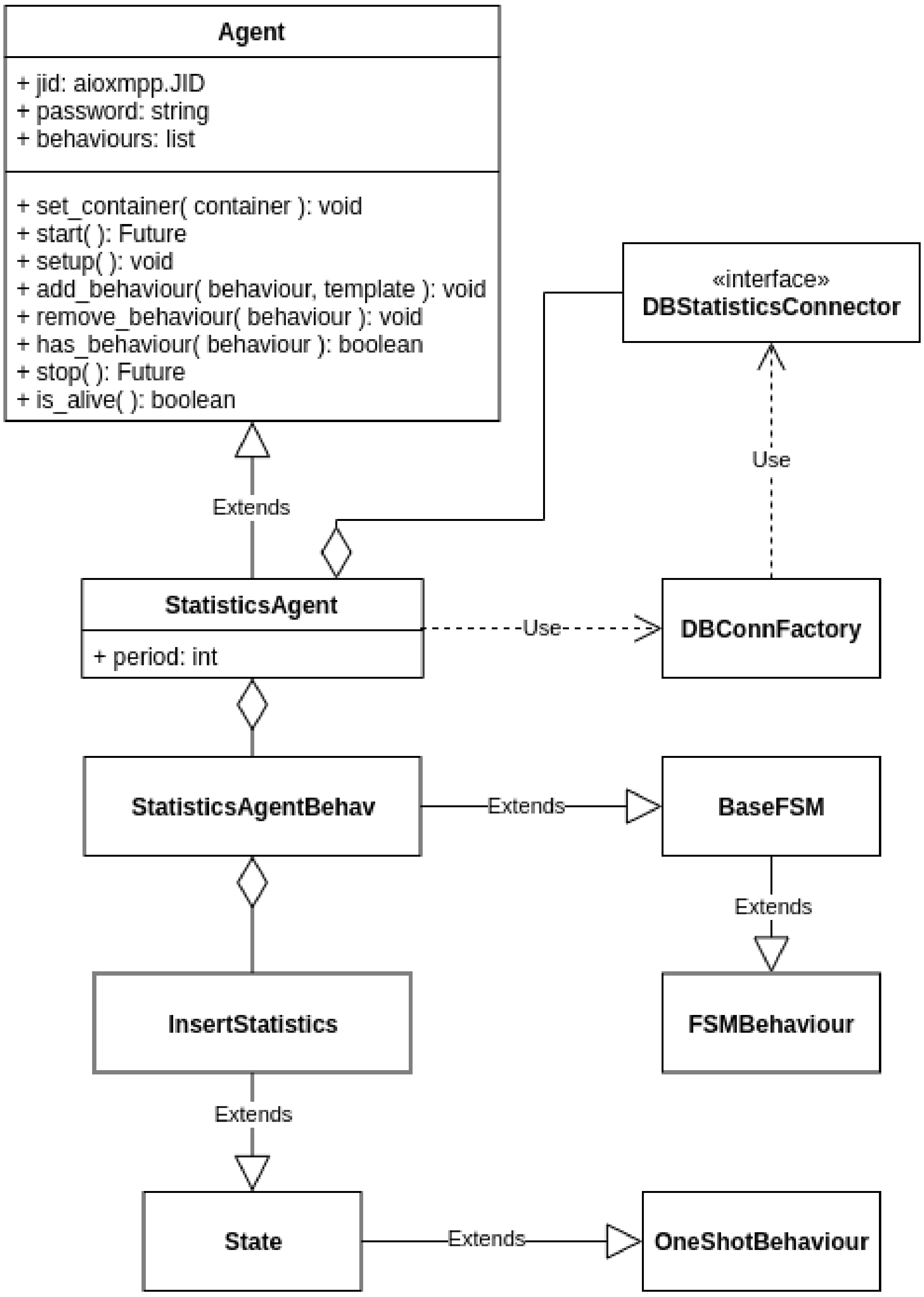


Figure 16: UML Class Diagram of StatisticsAgent entity

4.7 FrontEndAgent

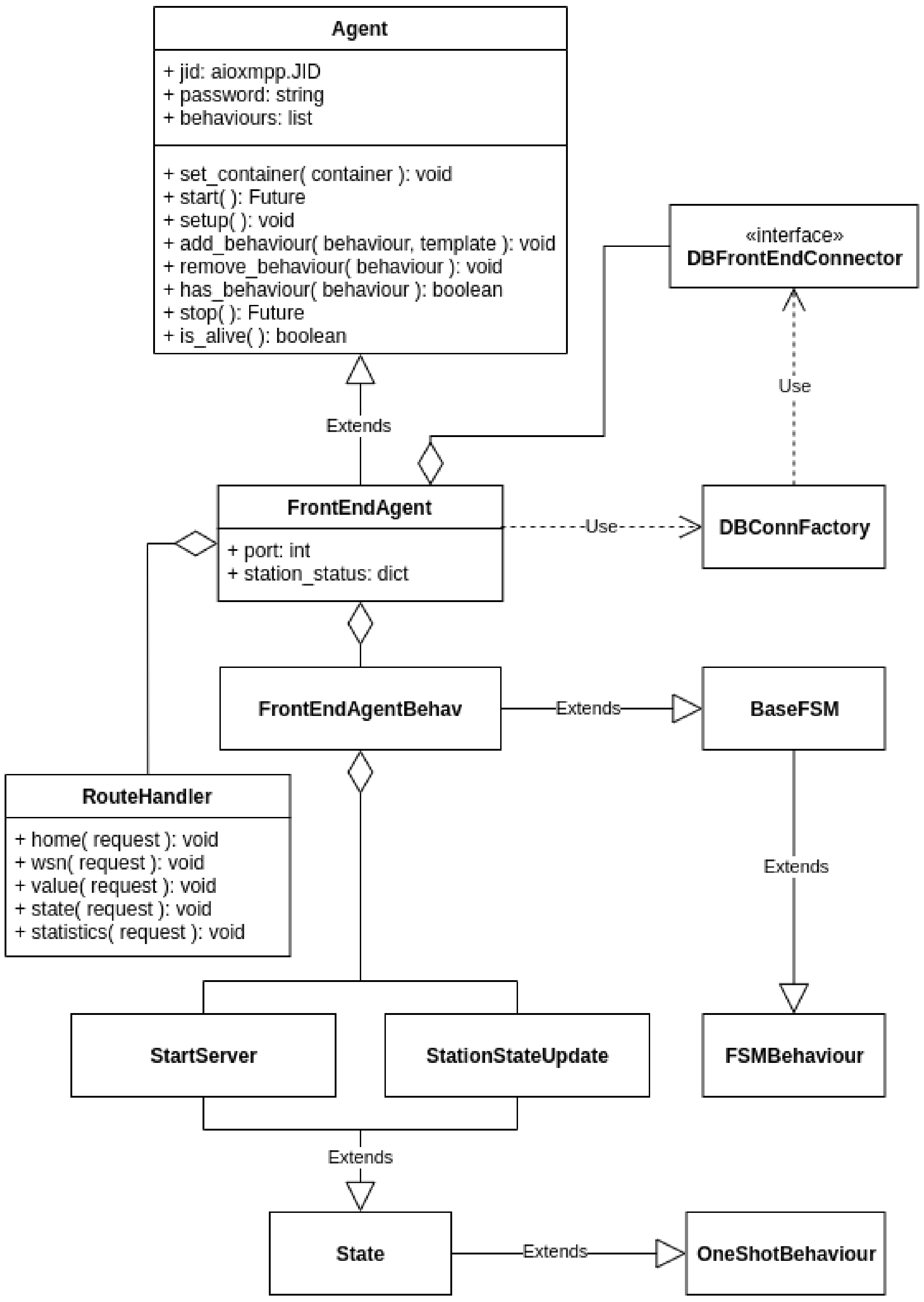


Figure 17: UML Class Diagram of FrontEndAgent entity

Each agent in SPADE provides a graphical interface by default that is accessible via web and can be activated by starting the agent's web module. This module can be used to create your own applications served by your agents themselves. You can register new paths in the web module and, following the MVC (Model-View-Controller) paradigm, register controllers that compute the necessary data from the agent (the model) and render a template (the view) which will be served when someone requests the path with which it was registered.

This functionality of SPADE allows to satisfy one of the non-functional requirements: to present data to the user through a web application.

The FrontEndAgent agent extends BaseFSM by defining two states: StartServer and StationStateUpdate. The StartServer state is responsible for launching the web application and the StationStateUpdate state is responsible for communicating with the TriggerAgent in order to receive real-time notifications about the status changes of SensorAgents.

To obtain the data and statistics relating to the SensorAgents it is necessary a specific database connector created through DBConnFactory.

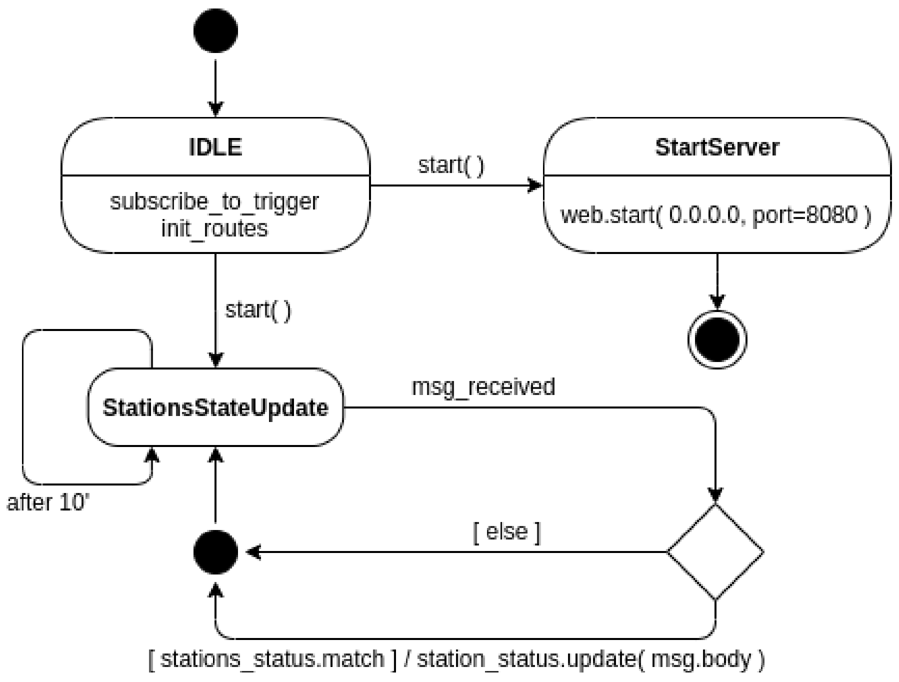


Figure 18: UML State Diagram of FrontEndAgent entity

4.7.1 Routes

The resources that must be available through the web service consist of the representation of the WSN, making the entire intermediate processing layer transparent to the user. The web service must provide the names that identify the SensorAgents, the values read by them and the related statistics such as the average, minimum and maximum values.

- GET /wsn, retrieves the list of jids of the SensorAgents that make up the network.
  - MIME Type: application/json
  - Responses:
    - 200, Success: an object is returned containing the jids of the SensorAgents.
    - 404, Not Found: there are no agents in the network.
- GET /value/{jid}, retrieves the detections made by the SensorAgent identified by the specified jid.
  - MIME Type: application/json
  - Query parameters:
    - limit (integer), optional parameter stating the exact amount of subresources to be returned. Default value: 1.
  - Responses:
    - 200, Success: an object is returned containing the detections.
    - 404, Not found: the provided jid does not exist.
    - 400, Bad Request: the provided jid and/or limit are invalid.
- GET /state/{jid}, retrieves the state of the SensorAgent identified by the specified jid.
  - MIME Type: application/json
  - Responses:
    - 200, Success: an object is returned containing the state of the SensorAgent.
    - 404, Not found: the provided jid does not exist.
- GET /statistics/{jid}, retrives the statistics made for the SensorAgent identified by the specified jid.
  - MIME Type: application/json
  - Query parameters:
    - limit (integer), optional parameter stating the exact amount of subresources to be returned. Default value: 1.

– Responses:
  * 200, Success: an object is returned containing the statistics made for the SensorAgent.
  * 404, Not found: the provided jid does not exist.
  * 400, Bad Request: the provided jid and/or limit are invalid.

4.7.2 Client Application

The main behavior of the client part of the web application, created with Javascript, consists of periodic scheduling function. The interaction with the server side, created through FrontEndAgent, starts by sending the HTTP request http://localhost:8080. Once the response is obtained, the customer takes care of obtaining the necessary data by sending requests on the specific previously defined paths and tracking the charts with them.

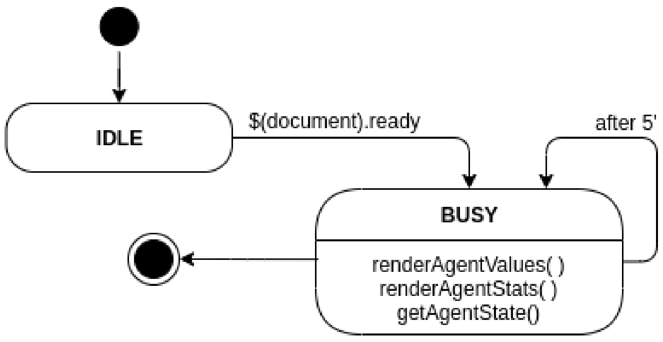


Figure 19: UML State Diagram of Client Web App

We have chosen to use the following Javascript libraries for the graphic aspect and for the dynamic rendering of the graphics:

- jQuery [4], library used to develop the Javascript code which consists of the client application.
- Materialize [5], responsive CSS framework based on Material Design by Google.
- Canvasjs [2], HTML5 charting library used to visualize the data by rendering graphs and charts on the screen.

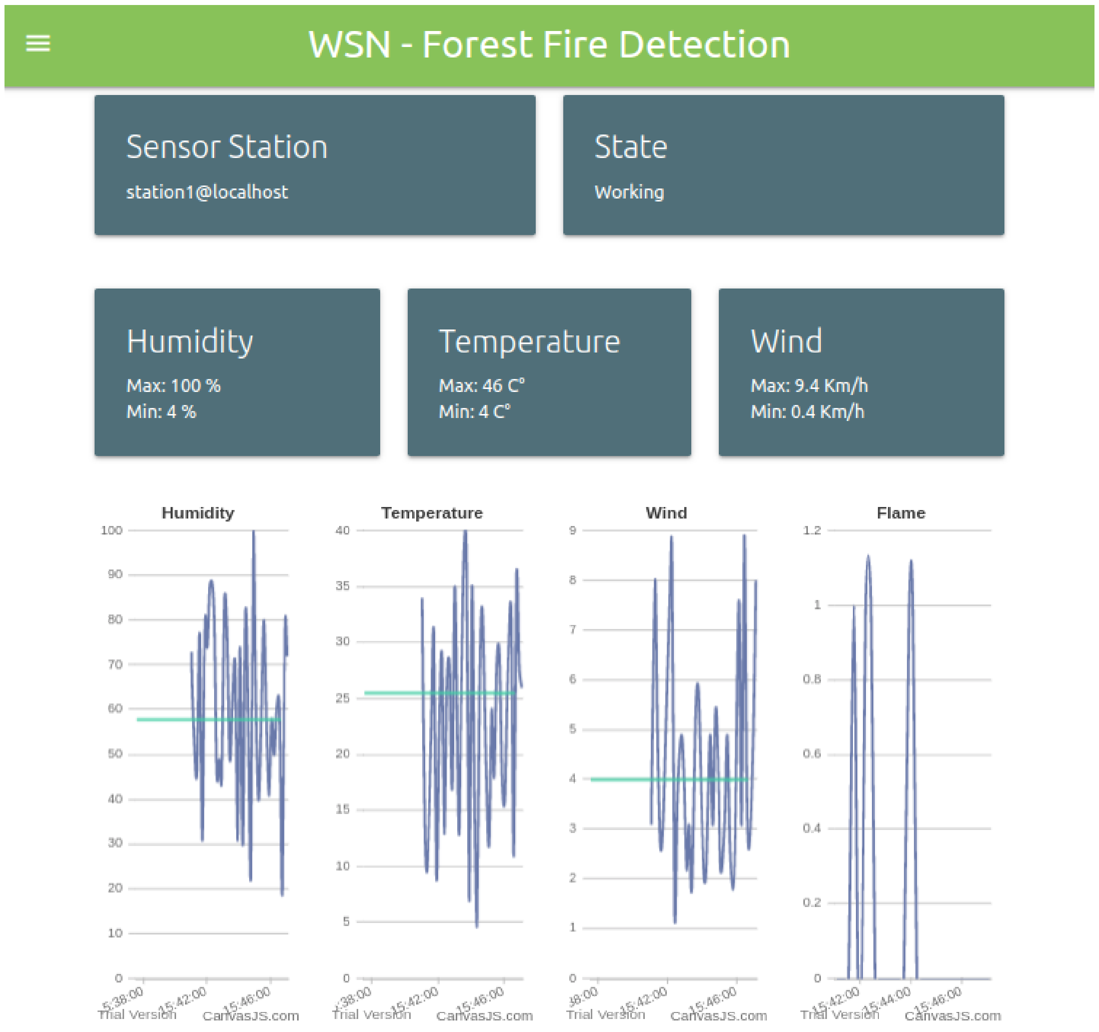


Figure 20: GUI screenshot of the web application

4.8 Interaction

Each SensorAgent entity obtains the values from its SensorStream and, without performing any processing on them, creates a JSON document which will become the body of the message to be sent to the DBManagerAgent. Before sending the message the performative metadata is set with the value "inform" in accordance with the FIPA specifications. The message sent by a SensorAgent to the DBManager has the following form:

```
<message to="dbmanager@localhost" from="station1@localhost"
thread="None" metadata={'performative': 'inform'}>
{"temp": 40.0, "hum": 41.0, "wind": 5.4, "flame": false}
</message>
```

Let event_name be the name of the value that quantifies the event to be measured: whenever this value does not respect a certain threshold, the agent TriggerAgent is informed by the agent DBManagerAgent by sending a message and he will check the situation and take decisions on the activation of the actuators [fig. 21]. It is necessary that the DBManagerAgent and TriggerAgent agents have the same knowledge on what is the name and type of the event to measure called event _name.

The DBManagerAgent agent informs the TriggerAgent agent only of the presence of an interesting new value and for this reason it uses the performative "inform" FIPA in the message. The body of the message contains the detection ObjectId and the agent_jid of the SensorAgent that carried it out:

```
<message to="trigger@localhost" from="dbmanager@localhost"
thread="None" metadata={'performative': 'inform'}>
{"_id_read": "5e4c0459a043d798fbf31a5b",
"agent_jid": "station6@localhost"}
</message>
```

Consider the most relevant case, that is, the correct detection of a fire. The TriggerAgent is aware of the jid of the SensorAgent which carried out the detection and indicates to it to perform the action on the actuators which in this case consists in activating the alarm. This is done by sending a message to the SensorAgent by using the FIPA performative "request":

```
<message to="station1@localhost" from="trigger@localhost"
thread="None" metadata={'performative': 'request'}>
{"alarm": true}
</message>
```

To inform the user in real time, both of the start and stop of the alarm, the TriggerAgent must inform the FrontEndAgent who will take care of displaying the information in an appropriate way.

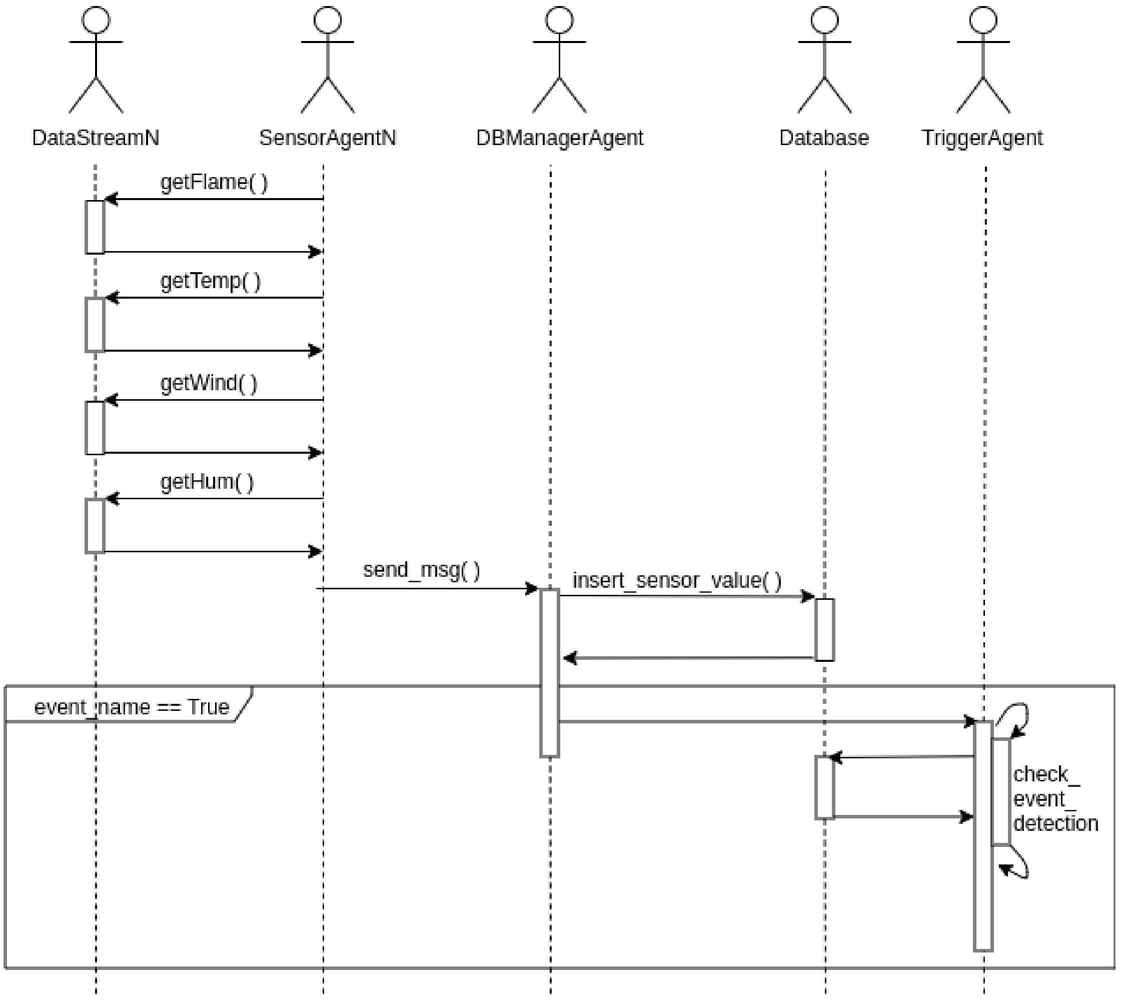


Figure 21: UML Sequence Diagram from the detection of environmental values to their control

The message that informs the FrontEndAgent of the start of the alarm has the following form:

```
<message to="frontend@localhost" from="trigger@localhost"
thread="None" metadata={'performative': 'inform'}>
{"station5@localhost": "Alarm On"}
</message>
```

The message that informs the FrontEndAgent of the alarm stop has the following form:

```
<message to="frontend@localhost" from="trigger@localhost"
thread="None" metadata={'performative': 'inform'}>
{"station5@localhost": "Working"}
</message>
```

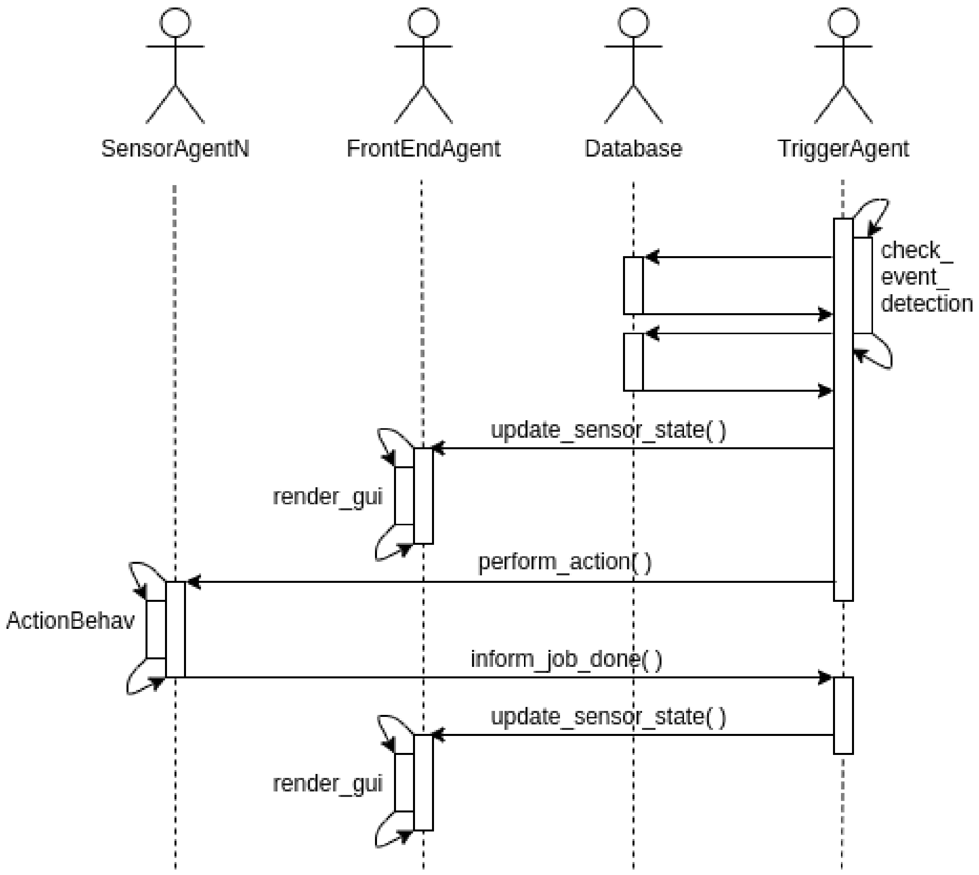


Figure 22: UML Sequence Diagram of the correct detection of a fire with the related signaling to the SensorAgent to perform the action on the actuators and to the FrontEndAgent to update the graphic interface

The message that informs the TriggerAgent that the SensorAgent has finished performing the action has the following form:

```
<message to="trigger@localhost" from="station5@localhost"
thread="None" metadata={'performative': 'inform'}>
"Job done!"
</message>
```

5 Spreading of Information

The spreading is based on direct communication among agents for progressively sending information over the system in order to increment the global knowledge. A copy of the information is sent to neighbours and propagated over the network from one node to another. Information spreads progressively over the system and reduces the lack of knowledge of the agents while keeping the constraint of local interaction. The different steps of the spreading process are:

- An agent initiates the spreading process
- The information spreads over the network
- The process finishes when information reaches all the nodes in the network

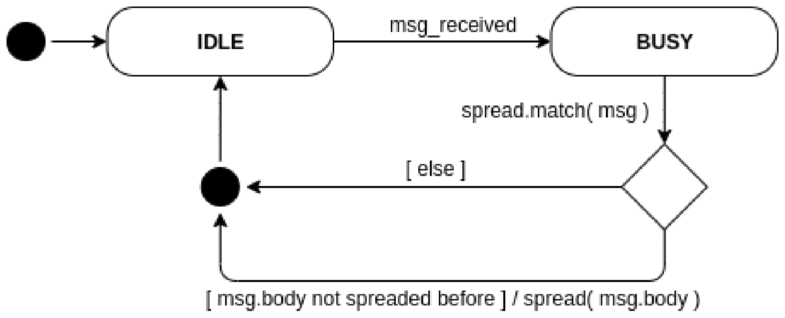


Figure 23: UML State Diagram of spreading process

If spreading occurs with high frequency, the information spreads quickly but the number of message increase causing a probable overload of the network. To reduce the number of messages, it is possible to specify the number of neighbouring agents to send that receive the information: it was demonstrated that it is not necessary to send the information to all the neighbouring nodes in order to ensure that every node has received the information.

We have chosen to manage the spreading by applying the behavioral design pattern Strategy in order to define different diffusion logics.

The implementation adopted in the Forest Fire Detection scenario, i.e. the entity SpreadingStrategyImpl, uses the *spread_param* parameter to determine the number of neighbors to send the information to:

- k = -1, the information is spreaded to all the neighbouring nodes
- k = 0, there is no spreading of information
- k > 0, the information is spreaded to k neighbouring nodes

In order to prevent the FFDsensorAgents from sending cyclically the same information between them, i.e. in a network structure modeled as a cyclic graph, it is necessary to take some precautions.

We have therefore chosen to store the information content of the message received and compare it with the messages received subsequently. By making these messages unique within the system through the use of the UUID (Universally Unique Identifier), it is possible to stop the diffusion once such specific information has been sent. To generate the UUID we decided to use the Python *uuid* library, which allows to generate random identifiers based on the MAC address of the machine and the timestamp.

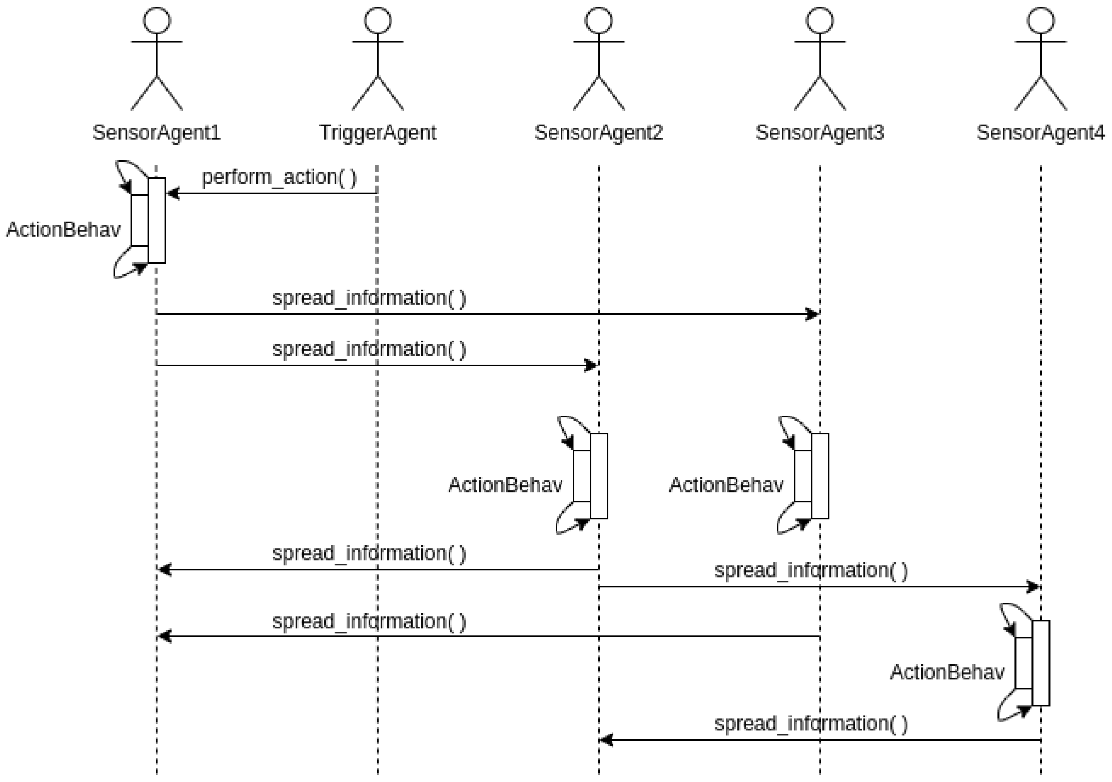


Figure 24: UML Sequence Diagram of spreading process

The spread of information is used in the Forest fire detection scenario to inform the other FFDSensorAgents about the detection of a fire and therefore to also activate the alarm on them. This is an example of applying spreading to the specific case of forest fire detection which allows in real applications to evacuate people present in the forest before they are involved in the fire.

Let $station1$ be an instance of FFDSensorAgent with $spread\_param = 2$. When $station1$ receives the alarm activation message from FFDTriggerAgent, it starts the spreading process by sending this information to two nearby random nodes via a message with the following form:

```
<message to="station2@localhost" from="station1@localhost"
thread="None" metadata={'performative': 'propagate'}>
"[6b22a17c-6f3a-11ea-aade-d8f2caa8a31d] station1@localhost
triggered the alarm"
</message>
```

When each FFDSensorAgent receives a message, with the FIPA performative "propagate" by another FFDSensorAgent on the network, it checks whether the information content has already been spreaded: in the negative case, it spreads the information to a number k of neighbors (where k is equal to its own *spread_param*), otherwise it will not spread the information further.

6 WSN Network Topologies

By definition a Wireless Sensor Network is a collection of nodes spatially distributed and organized in order to cooperate. Typically the organization of the nodes in the network, i.e. the topology, has three different roles with different characteristics:

- Coordinator, node responsible for forming the network, handling out address and managing the other functions that define the network.
- Router, node that can join existing networks, send information, receive information and route information.
- End Device, node that can simply send and receive information.

In the system created, the functions that characterize Coordinator and Router nodes are carried out by the infrastructure on which the Multi-Agent System is based. In this case using the XMPP protocol and the concepts of Agent Communication Channel, Agent Management System, Directory Facilitator and Agent Communication Language.

In classic topologies, such as Star, Mesh and Cluster Tree (Fig. 25), the End Device nodes communicate directly with each other logically but not physically: for this they make use of the Router and Coordinator nodes. The same occurs in the system created by exploiting the entities described above that provide the same functionality. This means that in order to have exactly the Router and Coordinator nodes belonging to the network it is necessary to define agents specifically.

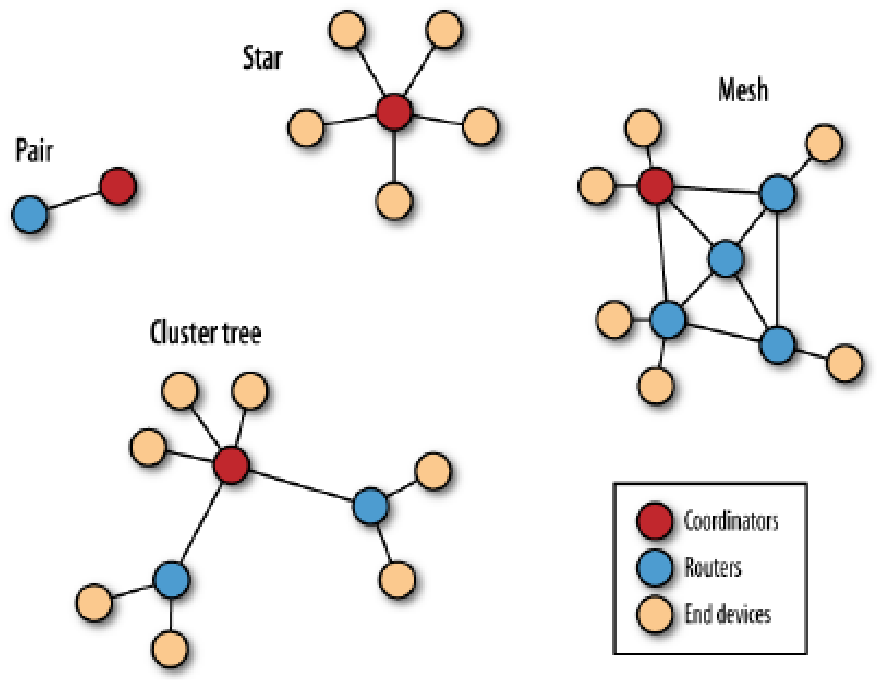


Figure 25: Classic Wireless Sensor Network Topologies

6.1 Building the network topologies

As previously mentioned, the structure of the Multi-Agent System favors the creation of the topology because it provides all the functions performed by the Router and Coordinator nodes. It is therefore necessary to focus on the protocol to be used to establish the logical connection between the End Device nodes.

Referring to the three-level architecture defined during the design phase, the agents that make up the WSN level are considered End Devices, i.e. the SensorAgent agents. The basic SensorAgent entities, which can be extended to create the specific SensorAgents of the various application scenarios, have the following features:

- given the jid of a specific agent, send a subscription request to that agent's contact list.
- automatically accept subscription requests received.
- respond to a subscription request with a subscription request if the agent jid is not already in the contact list.

Similarly to the features described above, we have defined features, in terms of unsubscription, for the removal of the connections between the SensorAgents.

The response with a subscription request to a subscription request allows you to automatically create bidirectional connections between the SensorAgents.

The effective creation of the network topology typically takes place at the first start of the system by carrying out an initialisation phase where the various End Devices send and accept the subscription requests defined by the network administrator.

The network topology is initialized at system startup but must be dynamic, that is, it must be able to be modified during its operation. The End Devices that characterize the nodes of a Wireless Sensor Network are in fact mobile entities. Typically they are not autonomous entities that move in space but can be subject to geographical shifts implied by many possible situations. Another situation that requires network dynamism is the ability to add or remove End Devices to the network.

Also in this case the characteristics provided by the Multi-Agent Systems and by the XMPP protocol allow to satisfy the needs described above.

6.2 Adopted Topology

In the system created, a network topology called 'Ring' was adopted, in order to carry out tests on information spreading in case an infinite loop can occur. In this topology every End Device has exactly two neighbors. The flow of information can follow both the clockwise and counterclockwise order as the connections between the End Devices are defined in a bidirectional way.

The specific topology adopted in the system includes five End Devices, which can be instantiated as follows:

```
station1 = SensorAgent("station1@localhost", "PWD")
station2 = SensorAgent("station2@localhost", "PWD")
station3 = SensorAgent("station3@localhost", "PWD")
station4 = SensorAgent("station4@localhost", "PWD")
station5 = SensorAgent("station5@localhost", "PWD")
```

The connections between the various End Devices are instantiated in the following way:

```
station1.presence.subscribe("station2@localhost")
station1.presence.subscribe("station5@localhost")
station2.presence.subscribe("station3@localhost")
station3.presence.subscribe("station4@localhost")
station4.presence.subscribe("station5@localhost")
```

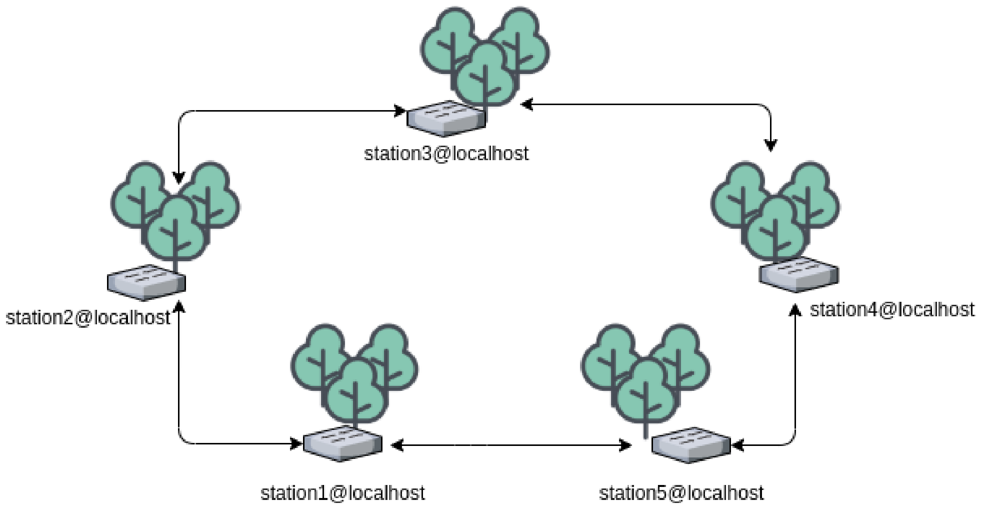


Figure 26: Ring topology adopted in the system scenario

7 Self-assessment / Validation

Several problems encountered in testing the developed agents led to the analysis of the tests developed by the developer of the SPADE framework Javi Palanca.

Initialization and deactivation of the agent container at the beginning and end of each developed test module allowed to solve various problems that occurred during the test. To do this, we used the related methods provided by *unittest*: $setUpClass(cls)$ and $tearDownClass(cls)$.

Another useful technique learned from the tests in SPADE is that of using the Mock concept to simulate certain events to which agents must react. For example, to test the reception on the FFDDBmanagerAgent side of the messages sent by the FFDSensorAgent it is not necessary to instantiate a new FFDSensorAgent but it is sufficient to define that single behavior through a MockedAgent and / or a MockedCoroutine.

In the self-assessment / validation phase we used the following Python libraries and frameworks:

- unittest, to develop the tests
- pytest, to run the tests developed
- pytest-asyncio, has additional features for testing asynchronous code
- pytest-aiohttp, has additional features for testing web servers created with the aiohttp library
- asynctest, has additional features for testing asynchronous code

The system is tested through various tests contained in specific modules: there is a module for each entity and for the related software components in order to verify its behavior in the specific scenario chosen for the project.

To run the tests you must first start the MongoDB server and the XMPP server:

```
sudo service mongod start
sudo service prosody start
```

And from inside the root folder of the project run the command:

```
pytest test/
```

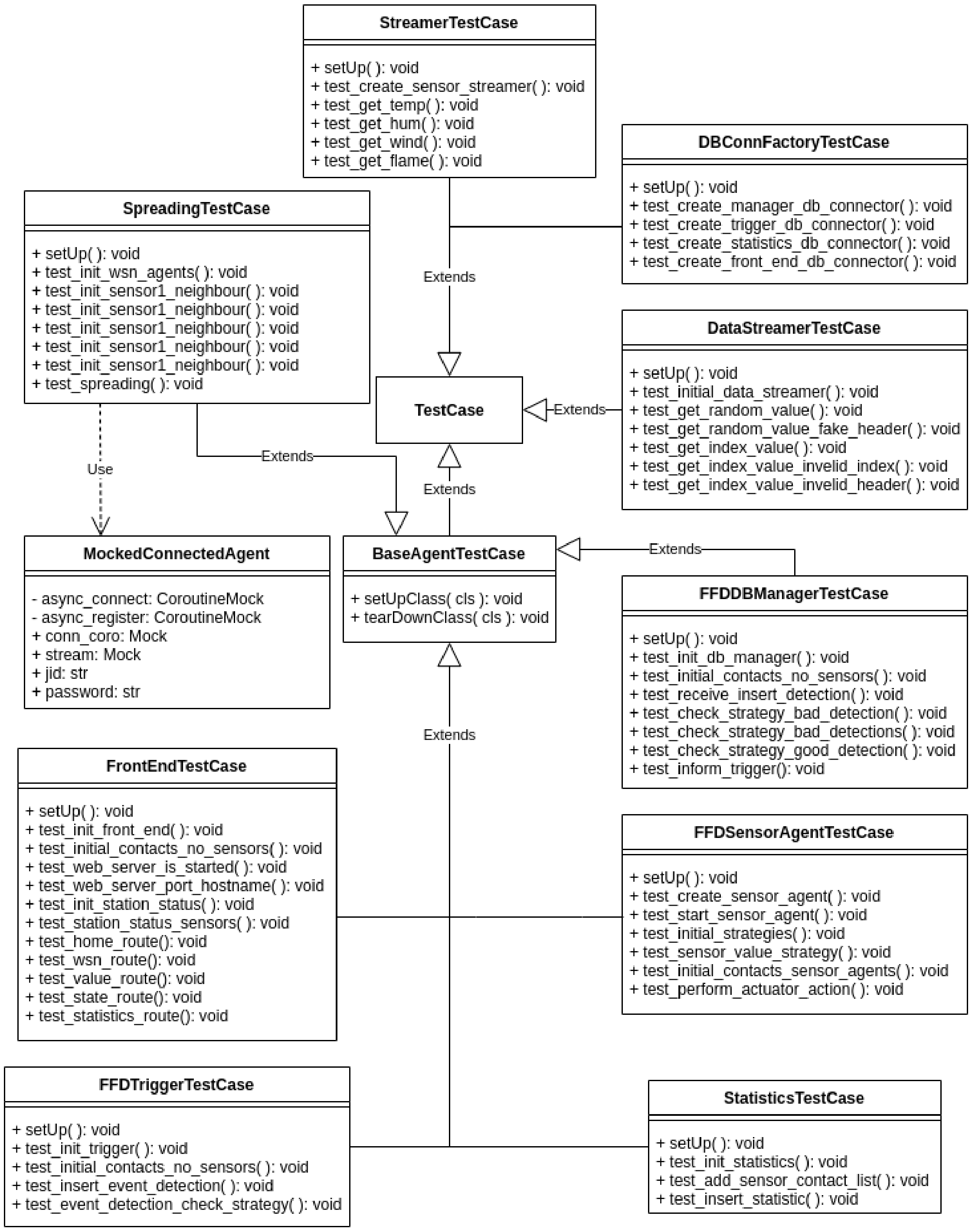

StreamerTestCase
+ setUp( ): void
+ test_create_sensor_streamer( ): void
+ test_get_temp( ): void
+ test_get_hum( ): void
+ test_get_wind( ): void
+ test_get_flame( ): void
DBConnFactoryTestCase
+ setUp( ): void
+ test_create_manager_db_connector( ): void
+ test_create_trigger_db_connector( ): void
+ test_create_statistics_db_connector( ): void
+ test_create_front_end_db_connector( ): void
SpreadingTestCase
+ setUp( ): void
+ test_init_wsn_agents( ): void
+ test_init_sensor1_neighbour( ): void
+ test_init_sensor1_neighbour( ): void
+ test_init_sensor1_neighbour( ): void
+ test_init_sensor1_neighbour( ): void
+ test_init_sensor1_neighbour( ): void
+ test_spreading( ): void
Extends
DataStreamerTestCase
+ setUp( ): void
+ test_initial_data_streamer( ): void
+ test_get_random_value( ): void
+ test_get_random_value_fake_header( ): void
+ test_get_index_value( ): void
+ test_get_index_value_invelid_index( ): void
+ test_get_index_value_invelid_header( ): void
TestCase
Extends
Use
Extends
Extends
MockedConnectedAgent
- async_connect: CoroutineMock
- async_register: CoroutineMock
+ conn_coro: Mock
+ stream: Mock
+ jid: str
+ password: str
BaseAgentTestCase
+ setUpClass(cls): void
+ tearDownClass(cls): void
Extends
FFDDBManagerTestCase
+ setUp( ): void
+ test_init_db_manager( ): void
+ test_initial_contacts_no_sensors( ): void
+ test_receive_insert_detection( ): void
+ test_check_strategy_bad_detection( ): void
+ test_check_strategy_bad_detections( ): void
+ test_check_strategy_good_detection( ): void
+ test_inform_trigger(): void
Extends
FrontEndTestCase
+ setUp( ): void
+ test_init_front_end( ): void
+ test_initial_contacts_no_sensors( ): void
+ test_web_server_is_started( ): void
+ test_web_server_port_hostname( ): void
+ test_init_station_status( ): void
+ test_station_status_sensors( ): void
+ test_home_route(): void
+ test_wsn_route(): void
+ test_value_route(): void
+ test_state_route(): void
+ test_statistics_route(): void
FFDSensorAgentTestCase
+ setUp( ): void
+ test_create_sensor_agent( ): void
+ test_start_sensor_agent( ): void
+ test_initial_strategies( ): void
+ test_sensor_value_strategy( ): void
+ test_initial_contacts_sensor_agents( ): void
+ test_perform_actuator_action( ): void
FFDTriggerTestCase
+ setUp( ): void
+ test_init_trigger( ): void
+ test_initial_contacts_no_sensors( ): void
+ test_insert_event_detection( ): void
+ test_event_detection_check_strategy( ): void
StatisticsTestCase
+ setUp( ): void
+ test_init_statistics( ): void
+ test_add_sensor_contact_list( ): void
+ test_insert_statistic( ): void

Figure 27: UML Class Diagram of test entities

8 Deployment Instructions

We designed and built the entire project on the Ubuntu 19.10 operating system and tested on the Ubuntu 19.10, Kali GNU/Linux Rolling and Linux Mint 19.3 operating systems and by using open-source software suitable for them.

The required Python version is 3.7.7 as with version 3.8 there have been problems with the web server created via aiohttp.

8.1 GitLab Repository

First you need to clone the repository `https://gitlab.com/pika-lab/courses/ds/projects/sd-project-paganelli-1920` by using the command:

```
git clone <repository>
```

Then you have to clone the repository and to install all the dependencies contained in the *requirements.txt* file, after checking that you have correctly installed the tools pip, setuptools and wheel on your machine, with the use of Package Installer for Python (PIP):

```
python -m pip install --upgrade pip setuptools wheel
pip3 install -r requirements.txt
```

After installing the project dependencies you need to install and configure the database and the XMPP server.

8.2 MongoDB Community Server

The version of MongoDB used in the project is the 4.2.3 for the Ubuntu 18.04 Linux x64 operating system. The *.deb* installation file can be downloaded at the url `https://repo.mongodb.org/apt/ubuntu/dists/bionic/mongodb-org/4.2/multiverse/binary-amd64/mongodb-org-server_4.2.3_amd64.deb`. Once the download is complete, you can start the installation by using the command:

```
sudo apt install ./mongodb-org-server_4.2.3_amd64.deb
```

To complete the installation, the system must be restarted.
There was no need to make any changes to the MongoDB settings, so the server will run locally on the default port (`mongodb://localhost:27017/`).

Optionally, if you want to use a graphical interface via web to view the data entered in the database, you can install the Mongoku [7] application via the command:

```
sudo npm install -g mongoku
```

This is a lighter alternative compared to the MongoDB Compass [6] application, which allows you to perform any operation on the database instead of just simple queries.

8.3 ProsodyIM

The version of the ProsodyIM XMPP server used in the project is 0.11.2-1 which is the one available via apt despite the recent release (2020-01-18) version 0.11.4. To install the XMPP server run the command:

```
sudo apt-get install prosody
```

The default configuration file *prosody.cfg.lua* allows the creation of a local XMPP server but does not allow automatic registration of XMPP clients. To enable this feature, it is necessary to modify this file. If it is not changed, you should use the *prosodyctl* command line application to manually add clients with administrator permissions. To modify the configuration file it is necessary to open it with administrator permissions:

```
sudo nano /etc/prosody/prosody.cfg.lua
```

The value of the *allow_registration* flag, which is false by default, must be changed to true.

```
allow_registration = true
```

After modifying the file you can quit nano by pressing $CTRL - X$ and confirming the saving of the modification by pressing y. Be careful not to change the file name.

9 Usage Examples

First of all you need to start MongoDB for the database server and ProsodyIM for the XMPP server. From a terminal run the following commands:

```
sudo service mongod start
sudo service prosody start
```

Check the correct execution of the servers using the commands:

```
sudo service mongod status
sudo service prosody status
```

If you have installed the Mongoku application, you must also start it using the command:

```
mongoku start
```

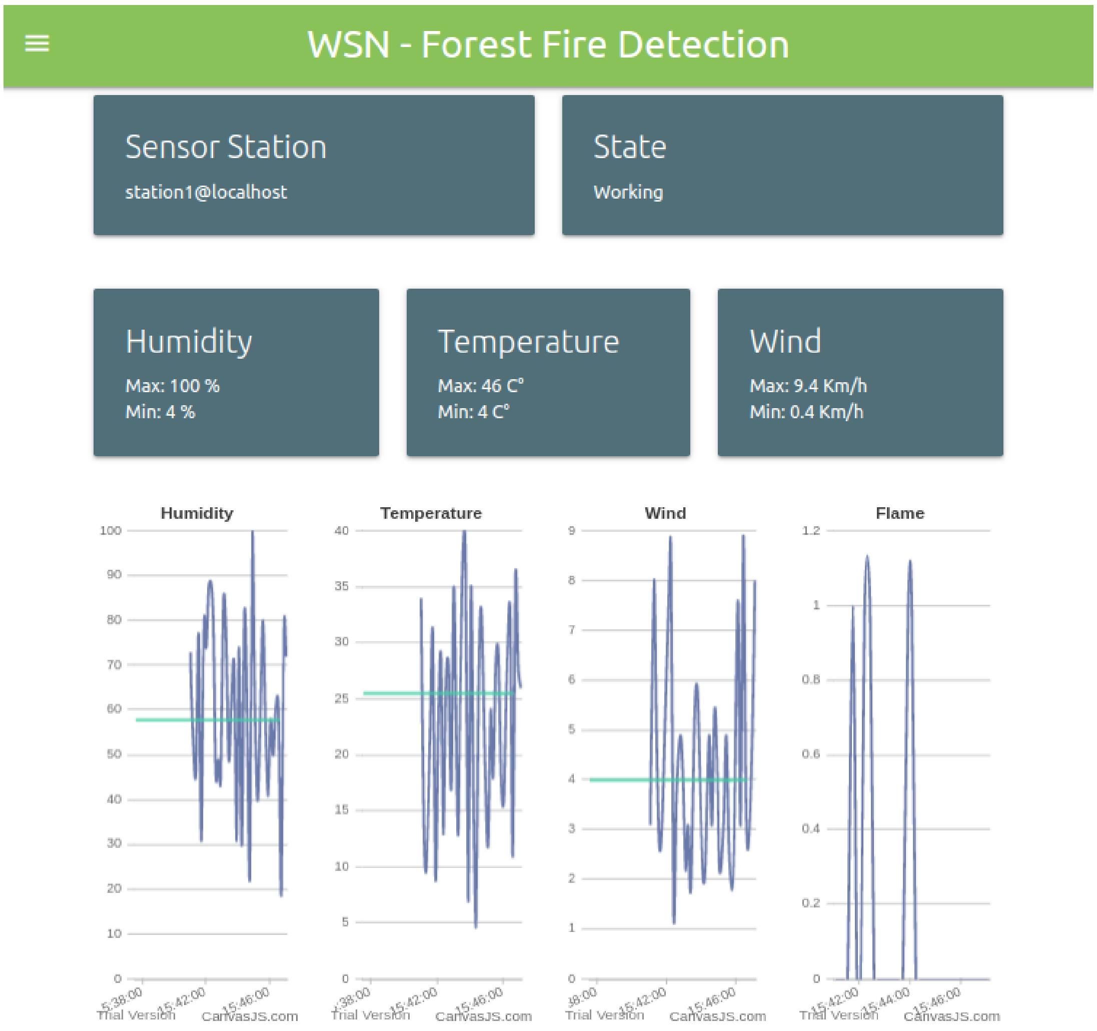
In this way it is possible to use the web client application provided by Mongoku by accessing the url `http://localhost:3100` from a browser.

At this point, from inside the root folder of the project, you can start the system by running the command:

```
python3 main.py
```

To connect to the data presentation server through the web application, you can access the url `http://localhost:8080` from a browser.

The main page (Fig. 28) is made up of four graphs that indicate the temporal trend of the individual measurements carried out by a specific SensorAgent but also the trend of the average. The three boxes above the graphs indicate the maximum and minimum values of the analog values measured by the SensorAgent. The two upper boxes indicate the jid of the SensorAgent whose detections and current status are being viewed. When a SensorAgent activates the alarm, the displayed status changes from "Working" to "Alarm-On" and vice versa when the alarm is deactivated.


WSN - Forest Fire Detection

Sensor Station
station1@localhost

State
Working

Humidity
Max: 100 %
Min: 4 %

Temperature
Max: 46 C°
Min: 4 C°

Wind
Max: 9.4 Km/h
Min: 0.4 Km/h

Figure 28: Home page of the web application

The button at the top left allows you to open the side menu containing the list of SensorAgents that make up the Wireless Sensor Network: selecting one of these the menu closes and all the data related to it are displayed (Fig. 29).

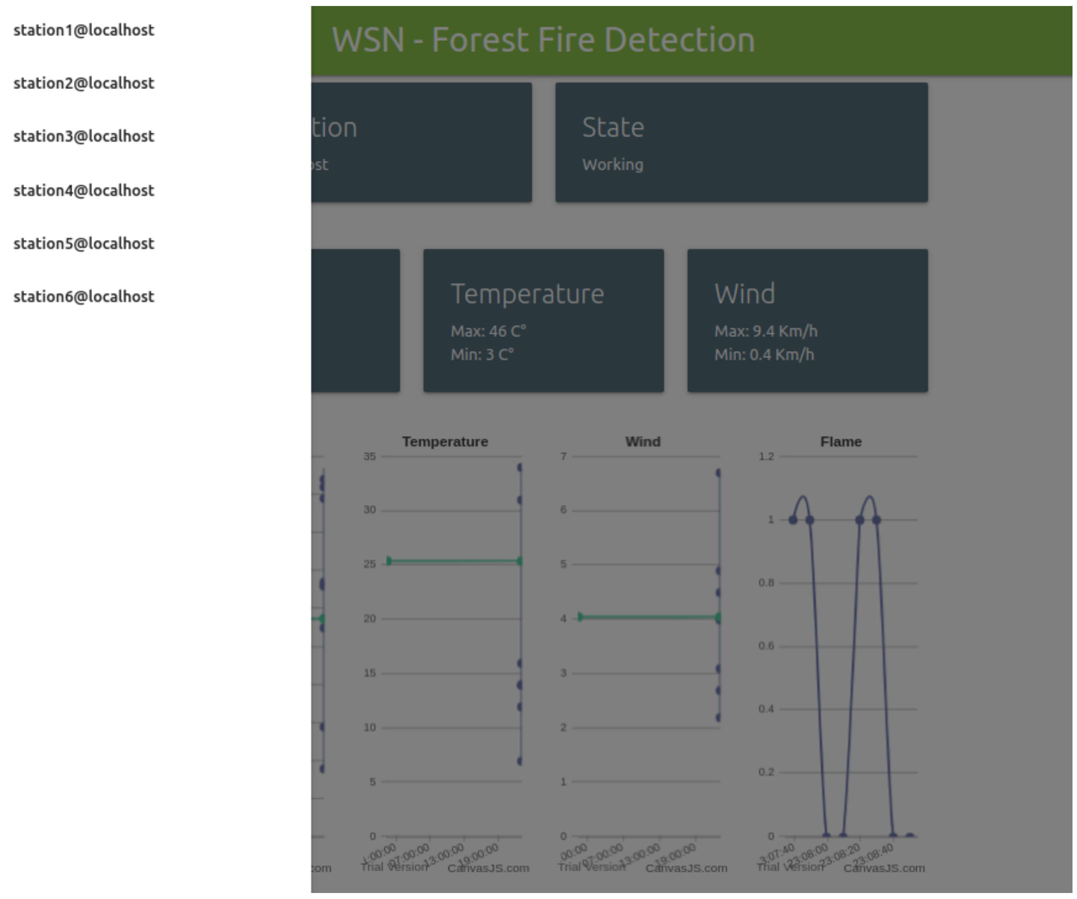


Figure 29: SensorAgent menu of the web application

10 Conclusions

The use of the agent programming paradigm for the management of a Wireless Sensors Network through the SPADE framework has made it possible to create a good starting point for the creation of a more complex and complete system.

The concept of a society of agents who collaborate with each other through interactions based on message exchange perfectly matches the typical use scenario of a Wireless Sensor Network.

In fact, we have successfully implemented the basic functions necessary for monitoring the environment and interacting with it, as well as those necessary for storing and processing the collected data and presenting them to the user.

The Forest Fires Detection, chosen as a case study for the use of the system created, allowed to verify the functioning of the system in a non-trivial way. Based on public datasets containing environmental values such as temperature, humidity and wind speed, the sensor readings have been simulated and all the necessary elaborations have been carried out in order to signal the presence of a fire to the user and to activate an actuator, also it simulated, to interact with the environment.

10.1 Future Works

First of all we would like to port the system to the JADE framework and to test the reusability of the project carried out by applying the system to a scenario different than the Forest Fires Detection.

In the system created there are some central points, such as the DBManagerAgent, which could be replaced by decentralizing also the storage of all the detections made by the SensorAgents.

Another future work consists in the introduction of agents that perform more complex elaborations on the data, compared to the statistics, by using machine learning techniques in order to make real predictions.

In addition to the aspects just mentioned, we would like to integrate the project with the future courses that we will attend, in particular with the "Applicazioni e Servizi Web" course and with the "Smart City e Tecnologie Mobili" course:

- With the "Applicazioni e Servizi Web" project, we would like to improve the entire web server used for presenting data to the user, also introducing additional features not currently present, such as activating and deactivating the alarm on a specific SensorAgent.
- With the "Smart City e Tecnologie Mobili" project we would like to create a real sensor station to be integrated with the system and in particular that can replace the simulated sensors without changing the code already developed. With this project we would also like to deploy the system on the network instead of locally using services such as those provided by Amazon Web Services or Microsoft Azure.

10.2 Personal Opinion

Overall, we are satisfied with the work done, in particular for having created a system that works in compliance with the requirements and approaching totally new technologies and programming paradigms for us.

The greatest difficulties during the development of the project were found in the use of the SPADE framework, in the related asynchronous programming techniques in Python but also in the testing of the latter. The main reason for the difficulties encountered in using the SPADE framework concern the lack of documentation. This inevitably led to spend a lot of time analyzing the framework code to understand how some aspects work, in particular the part that allows SPADE to use XMPP. Another reason, more personal, concerns the possibility of using SPADE on any XMPP server. We believe that there are obviously pros and cons in this opening of the framework but we also had to spend a lot of time in this to find an XMPP server with its configuration that would not give any problems.

References

[1] aiohttp library documentation. https://docs.aiohttp.org/en/stable/.

[2] Canvasjs. https://canvasjs.com/.

[3] Google dataset search - forest fire detection. https://datasetsearch.research.google.com/search?query=forest%20fire%20detection&docid=9OhchBsChyHfwjqPAAAAAA%3D%3D.

[4] jquery. https://jquery.com/.

[5] Materialize. https://materializecss.com/.

[6] Mongodb compass. https://www.mongodb.com/products/compass.

[7] Mongodb web client. https://github.com/huggingface/Mongoku.

[8] National wildfire coordinating group. https://www.nwcg.gov/.

[9] Prosody im. https://prosody.im.

[10] Pymongo documentation - python mongodb library. https://api.mongodb.com/python/current/.

[11] Pytest framework. https://docs.pytest.org/en/latest/.

[12] unittest framework. https://docs.python.org/3/library/unittest.html.

[13] Xmpp servers. https://xmpp.org/software/servers.html.

[14] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[15] Robert Faludi. *Building Wireless Sensor Networks: With ZigBee, XBee, Arduino, and Processing*. O'Reilly Media, Inc., 1st edition, 2010.

[16] J. Fernandez-Marquez, G. Di Marzo Serugendo, S. Montagna, M. Viroli, and J.L. Arcos. Description and composition of bio-inspired design patterns: A complete overview. *Natural Computing*, 2012.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[18] J. Palanca. *SPADE Documentation Release 3.1.4*. 2019.

[19] Divya Sharma, Sandeep Verma, and Kanika Sharma. *Network Topologies in Wireless Sensor Networks: A Review*. 2013.