University of Catania
Department of Mathematics and Computer Science
Doctoral Dissertation

Integration of distributed autonomous systems through a multi-layer, IoT-based hierachical approach

Massimiliano Maurizio De Benedetti

a dissertation
submitted to the department of
mathematics and computer science
of University of Catania
in fulfillment of the requirements
for the degree of
doctor of philosophy in computer science

Supervisor
Prof. Corrado Santoro, Ph.D.

Co-Supervisor
Dott. Fabrizio Messina, Ph.D.

Phd course in Mathematics and Computer Science – XXX cycle

A tutte le persone che mi hanno accompagnato in questo percorso.
Alla mia famiglia, per avermi sempre incoraggiato e supportato incondizionatamente.
A mio padre e mia madre, per avermi insegnato a non arrendermi mai e perseguire fino alla fine i miei obiettivi.
A mia sorella, per la forza ed abnegazione che ogni giorno dimostra e trasmette.
A Corrado e Fabrizio, per la grande professionalità, competenza ed affetto con cui mi hanno guidato in questo percorso.
A Fabio, per l'amicizia fraterna con la quale mi ha sempre supportato.
A mia moglie, per avermi dato la forza, la serenità ed il sostegno necessari a realizzare i miei obiettivi ed aver affrontato passo dopo passo ogni difficoltà al mio fianco.
Ad Arianna, perché possa trarre la forza di realizzare tutti i suoi obiettivi ispirandosi alla caparbietà e tenacia con cui ho cercato ed affrontato le sfide di questi anni.

Contents

List of Figures

List of Tables

Abstract

The huge number of IoT devices used nowadays for the various applications and environments introduces the problem of interoperability of heterogeneous platforms and software integration. Furthermore, the integration of heterogeneous platforms is complex and time consuming. In this disseration I tackle this problem by presenting the design of an approach aimed at providing a software platform called JarvSis. JarvSis is a distributed scheduler capable to automate the execution of multiple heterogeneous tasks on different applications by means of a modular and adaptable software architecture based on micro-services and nodes. JarvSis is capable to interact by design with any devices, from a complex robot platform to a simple sensor, that expose heterogeneous remote interfaces, e.g. web-api or MQTT message capabilities. The proposed approach is capable to automate the configuration and deploying of workflows of IoT related activities, by composing heterogeneous tasks organized in clusters, from the Cloud, to the Fog, to the smart devices running on the "ground". The tasks are organized in a hierarchical network on which Fog/Edge resources are used as a bridge between the computational resources hosted in the Cloud, and the devices operating on the "ground". In this configuration the top layers will provide control and coordination, while the bottom one is distributed among the Fog/Edge resources. As a proof-of-concept of JarvSis and its features, two detailed examples are provided in this dissertation: the former is related to the integration of IoT devices, the latter is related to robotic domain applications. In order to simulate the different scenarios and the *devices*, JarvSis has been coupled with a multi-agent simulator called AgentSimJs. AgentSimJs is a JavaScript simulation framework aimed at designing software simulations that can directly run on a Web browser. AgentSimJs is composed by a modular architecture made by a number of different modules to provide a set of flexible primitives to write all the parts of an agent behavior. Through AgentSimJs a simple and fast implementation of the aspects related to communication, motion, and group formation can be performed. Finally AgentSimJs includes the required capabilities to render a 3D scene with objects and agents, as well as to distribute the simulation among different machines and/or different threads.

Introduction

In the recent years we have assisted to a fast grow of the programming models, frameworks, middlewares and technological devices related to the Internet of Things (IoT). As stated in [68], IoT refers to the *networked interconnection of everyday objects, which are often equipped with ubiquitous intelligence. IoT will increase the ubiquity of the Internet by integrating every object for interaction via embedded systems, which leads to a highly distributed network of devices communicating with human beings as well as other devices.*

The relation among IoT and Cloud Computing is particularly interesting and multi faced: it involves several aspects like power capabilities, network latencies, multiple storage, and multi-tenancy [7]. The Cloud holds all the needed resources to support the analysis of data produced by IoT devices for medium-long terms decision. Nevertheless, the Internet of Things has introduced a few non-functional requirements as low-latency and geo-localization of devices and computational resources. All these requirements can be addressed by computational resources located at the "edge" of the networks, i.e. the Fog [6, 65]. Fog Computing represents a "reasonable" extension of the Cloud computing architecture: it is introduced to fill the existing gap between the IoT and the Cloud, and it is conceived to satisfy typical IoT constraints. The Fog layer does hold specific characteristics such as location awareness, low latency, geographical distribution of resources (vs. highly centralized Cloud resources), along with wireless access (i.e. mobility support), as well as real-time interaction, which are characteristics required to support IoT devices in their life-cycle [6].

Some challenges are still in place to support the development of distributed IoT applications. First of all, the adoption of Fog computing leads to the development of specific tools and methodologies for uniform management of the communication among different layer components, from the ground – IoT – , to the middle layer – the Fog – to the highly centralized Cloud resources. In particular, the analysis of several case studies highlighted the need of uniform controllers and interoperable message buses to be injected at the various levels to support the construction of suitable applications for any IoT ecosystem [6]. For instance, a distributed controller may analyze the state of the system, in order to change policies if needed, and to re-assign proper tasks to each subsystem. On the other hand, introducing such a support will lead to introduce a further problem of interoperability among the mobile, smart and autonomous entities which are placed in the ecosystem to perform the various tasks.

This dissertation focuses on the design and implementation of a highly modular software architecture to provide a distributed task scheduler named **JarvSis**, able to automate and coordinate the execution of myriads of heterogeneous tasks to execute in any IoT ecosystem [15].

First of all, JarvSis includes the specification of a simple uniform interface useful to trigger

heterogeneous tasks in "agnostic" way. This requirement is satisfied by simply relying on MQTT and JSON messages: in this way the user can integrate legacy applications by an additional simple wrapper for the specific tasks. As detailed in the following of this dissertation, JarvSis is able to interact with any kind of IoT device, as its components are suitable to be deployed also on devices with small resources.

Secondly, JarvSis provides a suitable support to organize the application as a hierarchical network with at least three layers: *(i)* the Cloud; *(ii)* the Fog, which acts as a bridge between the Cloud and the IoT; and *(iii)* a "ground layer", i.e. the JarvSis sub-network of heterogeneous, smart and mobile devices which represent the IoT layer. This characteristic will improve scalability of resources which are dedicated to interact with tasks and fault tolerance.

The contribution given by JarvSis in the design of a distributed IoT application is further discussed by illustrating two different case studies. The former is related to a realistic scenario concerning robot cooperation, which is needed in order to perform a number of tasks useful to reach a more complex task. We will show how a set of heterogeneous robots representing different technologies are easily integrated by JarvSis, which is a vertical integration platform. This case study is a proof-of-concept of the adoption of JarvSis as a standard integration methodology and platform due to its intrinsic integration capability of heterogeneous systems. In particular, using the JarvSis messaging system and the task aggregation and organization model, the different robots are able to communicate, exchange information and complete the several tasks. JarvSis can be used in a wide range of robotic applications. For instance, it is suitable to give support to the strategy for UAVs aerial mission detailed in [14].

The second case study concerns the typical IoT scenario that suffers of the the problem of IoT device integration: sensors or actuator produced by different vendors cannot be integrated easily and an ad hoc software layer must be developed accordingly to the particular integration needed. In this context it will be shown that JarvSis can be used to integrate such etherogeneous IoT devices by exploiting the MQTT protocol capabilities of the different devices. Using the standard message structure provided by JarvSis and the capability of propagating any payload between tasks and the hierarchical structure, a high number of IoT sensors linked to different software/platform and managed by different kind of gateways, can be integrated by JarvSis. In this case the slim implementation of JarvSis enables its execution in devices with limited computational capabilities (e.g. a raspberry pi-zero [30]).

Both case studies have been simulated by means of AgentSimJS [18], which is a Javascript-based solution to run 3D simulations of multi-agent systems in a web browser. As detailed in this dissertation, the design of AgentSimJs is highly modular, as a consequence code reuse can be properly addressed by sharing algorithms and behaviors developed by different users. In particular, two different components handle agent behavior and agent interaction, as well as the simulation of the physical environment and the agent-to-agent and agent-to-object collisions. In addition, the simulator offers the opportunity to distribute the simulation among several machines/thread

through the component named Web-API integrator.

This dissertation is structured as follows. Chapter 1 provides the necessary background and a discussion of related work. Chapter 2 illustrates the architecture and the functionalities of JarvSis as well as the main technological aspects involved with JarvSis. Chapter 3 contains a detailed description of the simulator AgentSimJS, which has been used to simulate the two case studies. The first case study is illustrated and discussed in Chapter 4, while Chapter 5 discusses the second case study. Finally Chapter 6 draws the conclusions with potential JarvSis enhancement and additional use-cases.

1

Background and Related Work

Device heterogeneity is a key aspect of IoT, because devices span from lightweight sensors to relatively powerful small computers, like smartphones. As a consequence, there is a need to design software solutions to support this heterogeneity in order to avoid redundant development, which would be very expensive. For this reason, schedulers, middlewares and frameworks for IoT applications are nowadays object of interests especially for companies that aim to invest money in the IoT market.

Among the current research, it is worth mentioning the development of an OS for IoT [2] having the aim of introducing a platform that can support devices with minimal resources as well as devices with more resources and capabilities. Such a solution allows the programmers to write C and C++ code with multi-threading and real-time capabilities with minimum hardware requirement (e.g. 1.5 kB of RAM).

Heterogeneity of IoT devices also affects network communication, because it gives very dynamic QoS requirements. Some researchers have studied this aspect and proposed a Continuous-Time Markov Chain (CTMC) traffic modelling, to coordinate the spectrum sharing framework for IoT [22]. They introduced a centralized scheduler in order to manage the different priorities of dynamic QoS requirements.

A different solution for scheduling tasks in an IoT context is presented in [67], which is designed to support video surveillance applications. In particular, a Cloud service is employed to support ubiquitous IoT nodes, and, due to limited capacity of each node, the authors designed an ad-hoc system architecture and a set of schedulers function and video processing algorithms. Although they have shown, by simulations, that their approach outperforms other scheduling methods, the approach which is presented in Chapter 2 is able to support any scheduling policies and algorithms. In particular the developer will have only to configure the scheduling policy on JarvSis, because the mechanisms for remote communications and interactions are already implemented.

A further interesting project is Obsidian [60], which is a Java-based task/job scheduler that allows system operators to schedule jobs at recurring times (like the Cron Daemon in the Unix OS [3]) along with a full UI for administrative tasks. One of the interesting feature of Obsidian is that it supports "zero-configuration" clustering to provide fail-over and job-sharing. Another interesting feature is represented by the support for event notifications which is very uncommon

in popular schedulers: notification capabilities deal with all the events related to the status of the jobs of the infrastructure, and are directed to the administrator of the infrastructure.

Quartz [62] is an open source job scheduling library which focuses on portability. Indeed it can be integrated in any Java application. In principle, Quartz can be installed in any IoT devices capable to run a JVM in order to schedule jobs in those devices. Another features is that it includes enterprise-class features, such as the support for JTA transactions [8] and clustering. JarvSis has been adapted in order to enable the communication with any IoT devices, and the construction of a hierarchical network of nodes providing a certain degree of resilience.

There are many proprietary and open source solutions for job scheduling and workflow execution on distributed systems. Some of them are very suitable for data maintenance and analysis on data centers [58] as well as load balancing of computational resources on small and medium companies. In the following we briefly discuss some of these solutions, as they include a number of characteristics that are similar to those described for JarvSis, then we describe the main differences.

Schedulix [34] is an Open Source scheduler for enterprises. It is the base version of the BIC suite [33], developed by the same company. It focuses on the design and integration of workflows of batch processes in enterprise IT infrastructures. Schedulix is equipped with a web front-end that allows the developer to design enterprise workflows by specifying hierarchies, dependencies, and pipelines among different jobs with the desired granularity. In Schedulix the concept of "External job" is mapped as a possibility to swap out sub-workflows to external systems in order to avoid the overloading of the system. Schedulix relies on a DBMS to store the information about the workflows, as well as runtime information about the processes. Its architecture is composed of three components, client, server and jobserver (which is deployed in the machines that will host the jobs of the workflow), and is compatible with Linux and Windows systems.

Another scheduler, designed with a philosophy similar to Schedulix, is Torque [12], an open source project based on the former project OpenPbs [61] originally developed the NASA Ames Research Center. Torque is composed by a server, which includes a configurable scheduler, a client to be installed into the machines which serve as user interfaces, and a component called "mom" to be installed into the computing resources in order to accept and execute the computing jobs. It is a mature project which incorporates advances in terms of scalability and reliability as well as a lot of functionalities, Torque does not include a workload manager but it can be integrated with MOAB, a product of the same company which is not free of charge, that is capable of placing workloads and adapting resources to improve the quality of service provided by the system.

OpenLava [56] is an Open Source Project fully compatible with IBM Platform LSF [37]. It is a workload manager that offers capabilities similar to those provided by Torque. Among others we can cite the possibility to specify dependencies for creating multi-step workflows, scheduling policy based on dynamic machine load flexible resource requirement syntax, and customizable job submission criteria. Morever, it provides Cloud support, i.e. application isolation, fast service deployment and cloud mobility by supporting deployment on Docker containers, and auto-scaling

support on Cloud resources, by easy adding or removing cluster nodes on the fly without cluster re-configuration.

The schedulers mentioned above provide excellent support for batch processing and workflow execution. On the other hand, the focus of JarvSis is partially different, as it includes the support to compose a hierarchical network with a high level of resilience. Moreover, JarvSis has been designed by focusing on the integration of modern technologies to deploy and connect IoT devices to Cloud and Fog resources.

2

JarvSis

As stated in the introduction of this dissertation, JarvSis represents a highly modular software architecture to provide a distributed task scheduler able to automate and coordinate the execution of myriads of heterogeneous tasks to be executed in any IoT ecosystem [15]. JarvSis is capable to manage a large number of heterogeneous tasks involved in a typical IoT scenario, for instance tasks that involve data collection from sensors or periodical sending of commands to devices for medical or military operations. Such tasks are often *grouped* on the basis of their domain and their correlations. JarvSis is able to handle this kind of applications thanks to its modular structure that features scalability and fault-tolerance, as described in the sections of this chapter.

2.1 Model

The underlying model of JarvSis represents a view of the typical IoT application as a set of *tasks* (logically grouped into *clusters*), that must be invoked according to certain specific policies.

In this work a task is intended as any *activity involving any kind of interaction among heterogeneous devices, which may be triggered due to different needs (check status of sensors, retrieving data and so on).*

These activities are performed by the JarvSis system which is designed to be executed on several *nodes* according to a schema that is detailed later in this chapter.

Let us consider, for example, a system composed of a number of sensors used to monitor the presence of smoke in an industrial plant. We suppose that such sensors are not passive but equipped with small processing capabilities[1] and thus they run a proper piece of software that performs environment monitoring. Data sampled by sensors are generally gathered in order to issue warnings and alarms, or simply for additional analysis. Based on the consideration above, we assume that sensors may expose an interface for the interaction with the external world and that a *gateway* is present in the system (a tiny Linux industrial PC), with the objective of executing the activities related to sensor polling, data acquisition, data storage, data-caching etc. In this context, in the gateway will run an instance of JarvSis properly configured to manage the monitoring application.

[1] e.g. a micro-controller with the needed firmware

As Figure 2.1 shows, the activities performed in the sensors are logically represented in JarvSis by a suitable entity named *functional task* (or simply *task*). Here, the role of JarvSis, with respect to the task, is to interact with the remote device in order to execute a number of activities, as for instance, checking if the device is alive, starting/stopping the device, executing a specific command in the device, retrieving data sampled by the device, storing data on the device, and so on.

Time, period and dependencies. According to its specific aim, each functional task is featured by the *type*, *period* and *dependencies*. The **type** can be either *one-shot*, meaning that the task has to be executed only one time, or *periodic*, that is, the task has to be executed on the basis of a certain **period**[2]. The **dependencies** specify relationships of a task with respect to other ones; a dependency means that a certain task can be executed only after the completion of another (or other) task(s); this may happen because the task could need data or actions performed by other tasks.

Clusters of tasks. In JarvSis, tasks can be grouped into logical entities called *clusters of tasks* or simply *clusters*. A *cluster* indicates a group of tasks logically or functionally correlated in order to run a specific goal (for instance the detection of the fire that must be performed by interacting with all the smoke sensors of the industrial plant). A cluster is featured by the tasks composing it and its **type** that, according to the nature of the cluster itself, can be classified in *parallel*, *sequential* or *hybrid*. In a *parallel* cluster, tasks can be executed concurrently, while a *sequential* type requires the tasks to be run one at time according to a sequence that is specified in the task dependencies; on the other hand, a *hybrid* cluster contains both parallel and sequential tasks, and their execution is governed, by JarvSis, on the basis of the specific dependencies. Like a single task, also a cluster may feature a **period**, meaning that the execution of the embedded tasks has to be run according to a specified time interval.

2.2 JarvSis Distribution and Hierarchy

Basically, a *JarvSis node* is a computational node that manages functional tasks grouped into clusters according to the definition above. Since, in large-scale systems, there may exist a very large number of IoT devices that have to be managed, and since each device has a JarvSis task as its representative, an architecture able to feature *scalability* is mandatory. Moreover, as it is usual in distributed applications, in order to avoid system stopping due to failures, *fault-tolerance* becomes another key aspect to we characterize the be taken into account.

JarvSis is aimed at satisfying the requirements correspondent to the aspects discussed above: indeed, a JarvSis system is composed of several computational nodes organized in layered tree-based architecture in which the level $k-1$ is responsible of controlling level k, as it is shown in Figures 2.2. The lowest level (n in Figure 2.2) is in charge of executing the tasks related to

[2]Here the period is intended to be not critical, thus it is treated in a best-effort way.

the interaction with devices; the distribution of IoT tasks among different nodes, which can be performed either manually (by means of configuration files) or managed by JarvSis itself through a proper algorithm 2.6, ensures the proper scalability. Fault-tolerance is instead handled by performing a monitoring of the nodes of level n; indeed, since tasks are grouped into clusters, we consider each cluster of level n as a *task* of the level $n-1$ which, in turn, is composed of other JarvSis nodes: in this way, level $n-1$ can control n by not only triggering cluster execution but also checking if it is working properly. This dependency between levels is repeated recursively till reaching the top-level in which a single JarvSis node is in charge of managing the entire system.

Management operations performed by level $k-1$ on level k are not only related to checking if the node is alive but also to ensure that the workload of underlying nodes is not as high as to cause performance problems. Indeed, as it will be detailed in Section 2.6, if level $k-1$ finds a problem in a node of level k (e.g. high load or node fault), it can decide to migrate the relevant tasks/clusters to another node of the same level thus ensuring the system can continue to operate correctly.

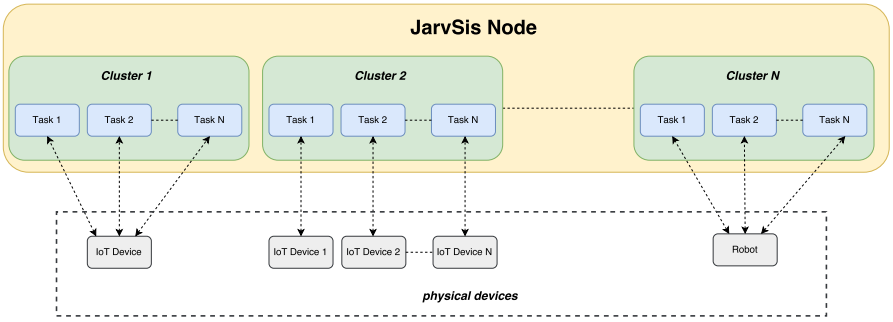

Figure 2.1: JarvSis: Node, Clusters and Tasks

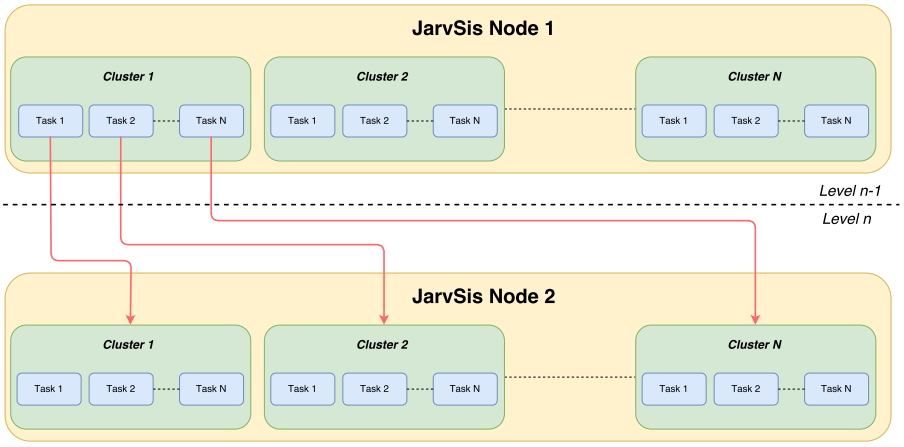

Figure 2.2: JarvSis: mapping clusters to tasks of the upper layer

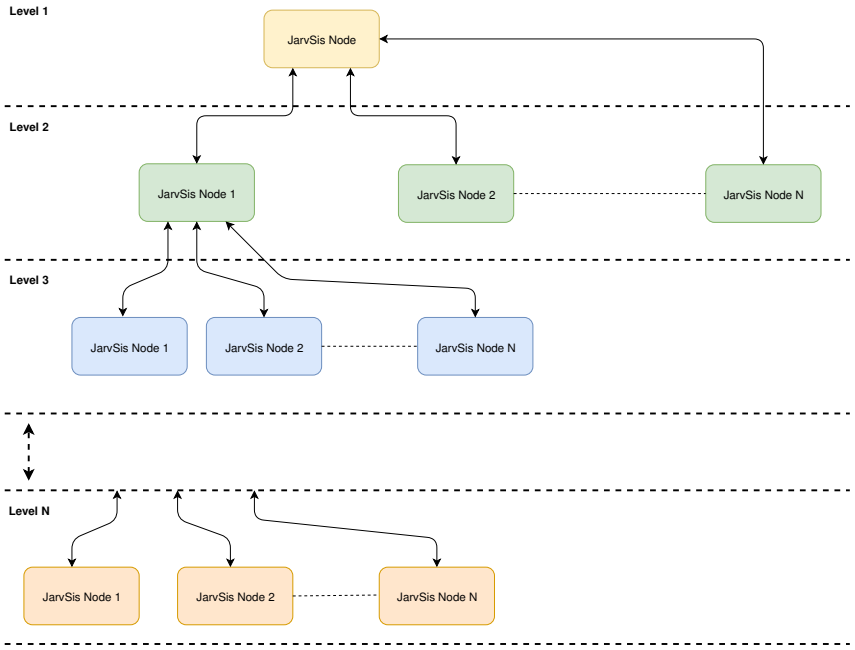

Figure 2.3: JarvSis layered structure

2.3 Architecture

The architecture of JarvSis is composed of three main blocks: **Execution**, **Control** and **Connection**. In turn, each block is made by one or more components, as depicted in Figure 2.4.

The block **Execution** consists of the components *Adaptive Scheduler* and *Multi-thread controller*, which handle the functionalities to manage the execution of clusters of tasks. The block **Control** is represented by the *Manager*, which is capable to manage all the entities of the execution layer by checking their status, the overall workload and, as a consequence, by performing balancing operation when needed. The block **Connection** is made by the *Client APIs* and an *MQTT Client*, eventually exploited by a web front-end, in order to provide communication capabilities with the Control layer, and by the *Data Mapping & Entity Connection*.

The **Multi-thread Controller** is the component in charge of concretely executing the tasks; it is essentially made of a *thread pool* and each thread is assigned to a specific task (it will be used at runtime to launch the related task); upon task completion, the thread becomes “free” to host a new task. In case of ”critical” task that can be executed when an external message is received the thread is permanently allocated to the task.

The **Adaptive Scheduler** is the high-level governor of the clusters/tasks. It is the entity that decides the order of execution of clusters and, as a consequence, tasks; such decisions are made surely on the basis of the constraints imposed in task and cluster specifications, i.e. periodicity, task dependencies, cluster type, but also according to other aspects like resource availability or overload conditions. Indeed, it is the Adaptive Scheduler that hosts the needed “intelligence” to ensure that resources allocated for the clusters are not overloaded; it can split a cluster in two or more clusters

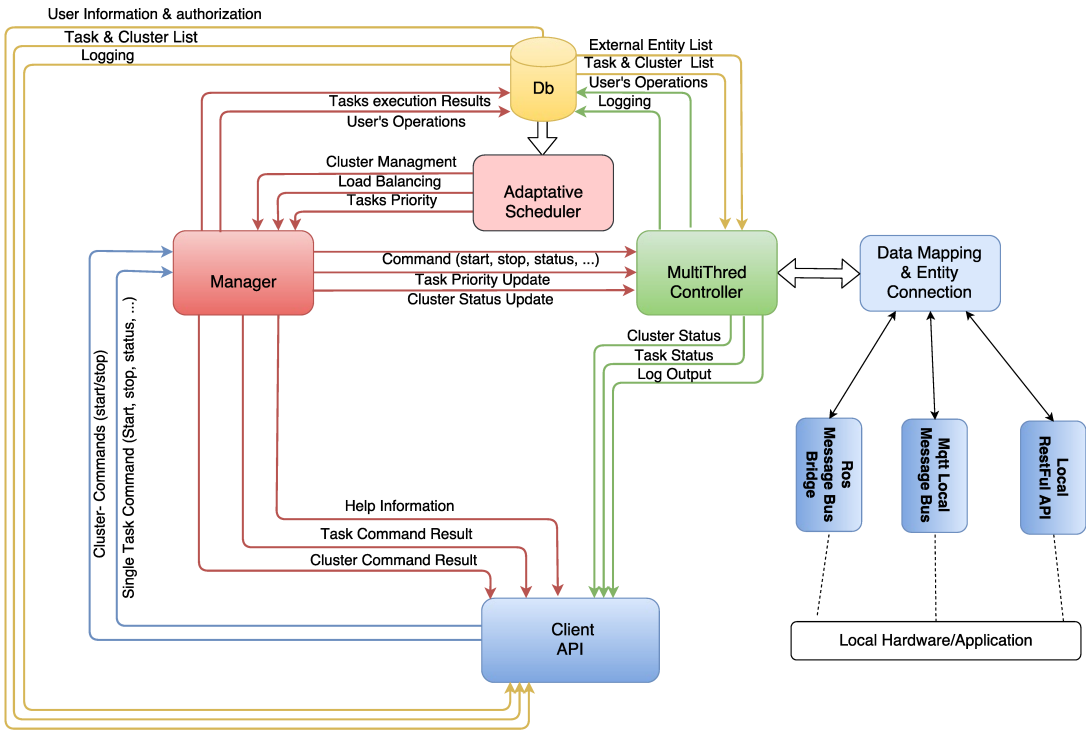

Figure 2.4: JarvSis: components and interactions

in order to give more resources to the tasks, obviously given that the operation is compliant with tasks mutual dependencies; or it can move a given cluster to another node, for load balancing purposes. The current implementation Adaptive Scheduler is focused on cluster/tasks switching but it can be customized by the user that can implement custom clusters/tasks management algorithms through the Adaptive Scheduler interface. This interface provides all the methods that must be used to modify/implements/migrate a cluster/tasks within a JarvSis network.

The **Manager** acts as a "mediator" between the Adaptive Scheduler and the Multi-thread Controller. It is informed from the former component about the tasks to be executed and their execution strategy and, in turn, sends directives to the Multi-thread Controller in order to concretely trigger task execution. The Manager is also able to communicate with the JarvSis Client API in order to let the external world interact with the JarvSis node.

The **Data Mapping & Entity Connection** is responsible of performing interaction between a JarvSis node and an IoT device, as well as between JarvSis nodes. It also manages data mapping operations to translate the data coming from external entities in a format suitable for the internal components. Indeed, as discussed later, the interactions in a JarvSis network are implemented through the message bus technologies *MQTT* [4] (also used for internal JarvSis node communication) or by means of a RESTful WebAPI.

The **Client API** is responsible of exposing a RestFul API that can be used from the Web front-end or an external application (eg. integration of different cloud application that expose

their own RestFul API).

Finally, the architecture relies on a dedicated Sqlite database (the **Db** component) used to manage the tasks, clusters, users information and authentication.

URI	Method	POST body	Result
startTask	POST	JSON payload	Start the task with the given payload in JSON format
stopTask	GET	empty	Stop the execution of the task
status	GET	empty	Retrieve the status of the task
result	GET	empty	Retrieve the result of the last execution of the task
getErr	GET	empty	Get the details of the last execution error

Table 2.1: RestFulWeb-API of a JarvSis task

2.4 Communications

In order to enable the interaction with IoT devices, JarvSis is able to rely on two different technologies: *MQTT* or *RESTful WebAPI*. MQTT is a standard "machine-to-machine" protocol suitably designed to let IoT device communicate and is based on TCP/IP. The RESTful WebAPI is instead based on interactions that exploits the standard HTTP protocol.

In order to connect an IoT device to a JarvSis network, the device itself must expose a number of services compliant to the RESTful Web-API described in Table 2.1, or in the form of a MQTT broker capable to handle a similar set of "methods" as those described in Table 2.1. In this way, different heterogeneous tasks hosted by different devices can be managed by JarvSis transparently, provided that the developer has written the proper code to implement the Web-API interface or the communication through MQTT.

JarvSis takes information about the interface of the tasks by means of the URLs specified in the configuration files that must be provided in the JarvSis node. These configuration files include the XML definition of the tasks to be managed by the node clusters by indicating the nature of the tasks and the dependencies among them. As the example in Listing 2.1 shows, where a cluster of six tasks organized in a workflow is specified, the attribute `endpoint` is used to specify the URL that, in turn, indicates the protocol type (HTTP for tasks from $t1$ to $t5$, MQTT for task $t6$), while dependencies are expressed by means of elements `<in>` and `<out>`.

2.5 Construction of a JarvSis network

A JarvSis network is constructed incrementally by plugging the needed nodes by means of a simple configuration. The process leading to plugging of a new node in the JarvSis network is explained below. First of all, a skeleton of the configuration must be provided by means of an XML file (see Listing 2.2) containing only the address of the parent, which must be a valid address for an MQTT

Listing 2.1: JarvSis Cluster Specification

```
"clusters": [
   {
     "cluster_name": "single_drone_insp",
     "ext_application_id": "1",
     "linked_to_parent_node_task": "0",
     "ext_cluster_id": "1",
     "cluster_description": "cluster used of single drone management",
     "parent_node_task": "?",
     "cluster_type": "seq_cluster",
     "ext_application_name": "agentsimjs",
     "task_array": [
       {
         "task_name": "task_seq_1",
         "task_description": "dummy_task",
         "ext_application_id": "1",
         "task_priority": 1,
         "task_starting_type": "sequential",
         "ext_task_id": "1",
         "task_timeout": "1000",
         "ext_application_name": "sample_application",
         "task_type": "external",
         "child_node_cluster": "?",
         "task_timer": "?",
         "linked_to_child_node_cluster": "0"
         "endpoint="http://jarvnet.dmi.unict.it:2000/t1_jarvnet_dmi_unict_it"},
       {
         "task_name": "task_seq_2",
         "task_description": "dummy_task",
         "ext_application_id": "2",
         "task_priority": 2,
         "task_starting_type": "sequential",
         "ext_task_id": "2",
         "task_timeout": "1000",
         "ext_application_name": "sample_application",
         "task_type": "external",
         "child_node_cluster": "?",
         "task_timer": "?",
         "linked_to_child_node_cluster": "0"
         "endpoint="http://jarvnet.dmi.unict.it:2000/t2_jarvnet_dmi_unict_it"},
       {
         "task_name": "task_seq_0",
         "task_description": "dummy_task",
         "ext_application_id": "1",
         "task_priority": 3,
         "task_starting_type": "sequential",
         "ext_task_id": "3",
         "task_timeout": "1000",
         "ext_application_name": "sample_application",
         "task_type": "external",
         "child_node_cluster": "?",
         "task_timer": "?",
         "linked_to_child_node_cluster": "0"
         "endpoint="http://jarvnet.dmi.unict.it:2000/t3_jarvnet_dmi_unict_it"
         }
       ]
     }
   ]
```

broker running in the parent node. Then, once the new node is up, it contacts its own parent by means of the MQTT [4] protocol, the specified address and a standard endpoint, e.g. "`/join`", in order to ask the parent to assign it an Id and to create a specific MQTT endpoint which will represents the communication channel between the new node and the parent. The parent will send the information about the Id to assign to the child and the MQTT endpoint to its new child, by means of a JSON message [38]. A simple version of such a JSON message is given in Listing 2.3.

Listing 2.2: JarvSis Node configuration

```
<configuration>
  <node_definiton_parameters>
    <level></level>
    <node_id></node_id>
  </node_definiton_parameters>
    <log_configuration>
      <log_type>dual</log_type>
      <log_mode>verbose</log_mode>
    </log_configuration>
    <parent_node_config>
      <has_parent_node>1</has_parent_node>
      <parent_node_id></parent_node_id>
      <parent_broker>"tcp://px.jarvnet.dmi.unict.it:1883";</parent_broker>
    </parent_node_config>
</configuration>
```

Listing 2.3: Response of a JarvSis node (the parent) to a new node that wish to join (the child)

```
{
        "msg_type":"first_connection_request", //msg type -> request connection to parent node
                "node_id": "127679335911650", // sender node id
                "target_node_id": "127679345911850", // target parent node id
                "node_global_level":"?", // sender node global level
                "message_sent_time", "" // sent time of the request
}
```

To establish a connection with a parent node, each JarvSis node runs a specific connection request algorithm (2.5). The startup procedure of a JarvSis node as first step will execute the connection request algorithm that is composed by:

1. preliminary check of the node global level within the JarvSis network (the global node level can be defined by the user but it will be checked by the parent node).
2. check if the Parent Node is defined (the Parent Node Id is defined)
3. check if the Parent Node is connected (there is the confirmation message saved on local database).
4. Send a connection request to the parent node if the parent node is defined but not connected.
5. start the JarvSis node accordingly to the previous step.

To receive all the connection request message (and other generic message related to the JarvSis network monitoring) each JarvSis node creates a specific topic called "generic_comm". This topic is initially used by all the child nodes that want to connect to a specific parent node to send a connection request message (2.6).

If the response of the parent JarvSis node to the connection request is positive then a dedicated topic is created by the parent node. To finalize this process the child node must send a final

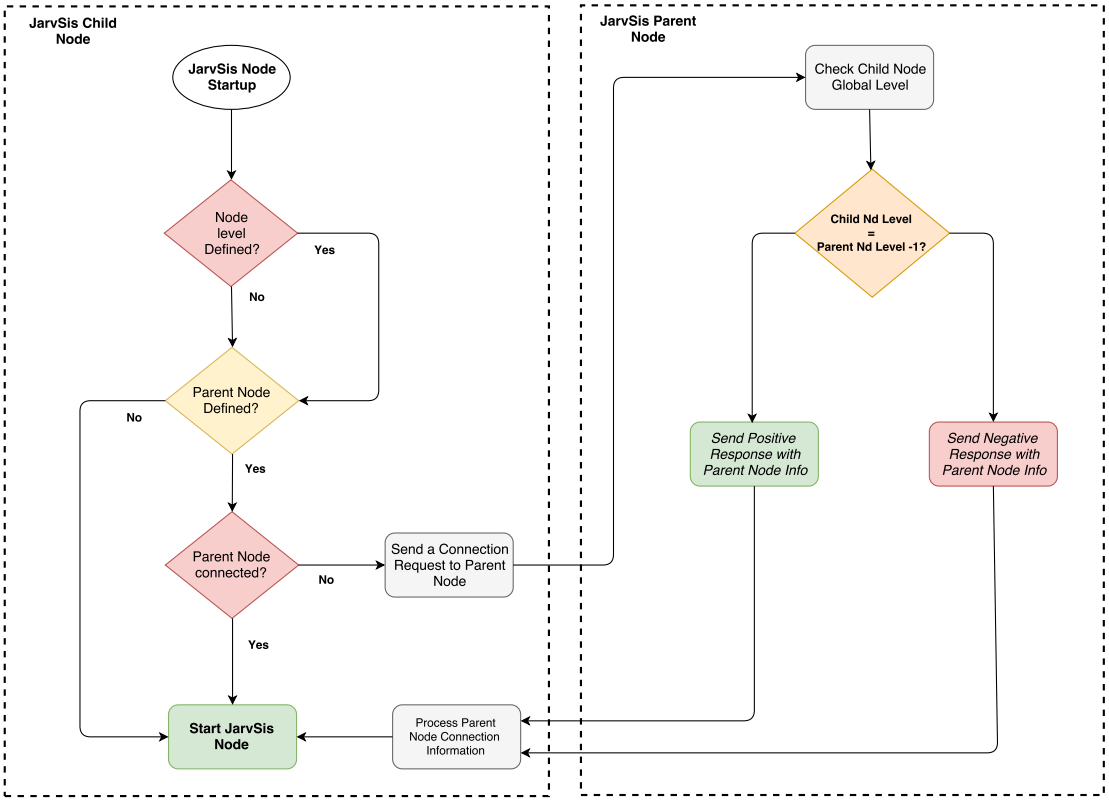

Figure 2.5: JarvSis Node First Connection Algorithm

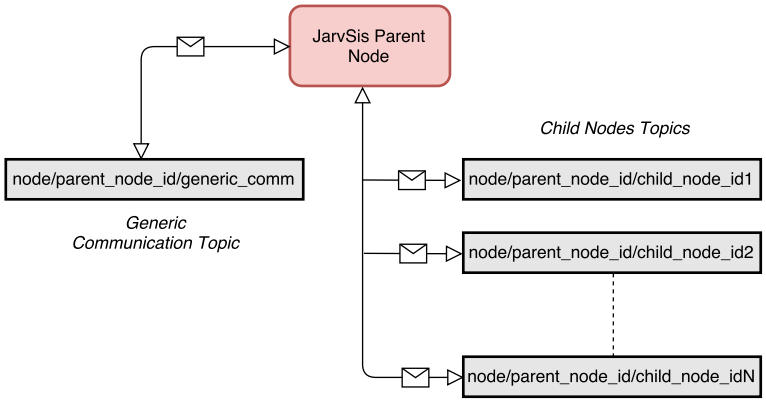

Figure 2.6: JarvSis Parent Node Topics

confirmation message to the parent node in this topic. This dedicated topic will be used by parent and child node to all the communication message related to the cluster/task and child node management (2.7).

2.6 Resilience, load balancing and scalability

In a JarvSis network, each node is able to communicate with its peers. In particular, each node is able to communicate with *(i)* its own *child* nodes (it can send directives), i.e. all the nodes directly

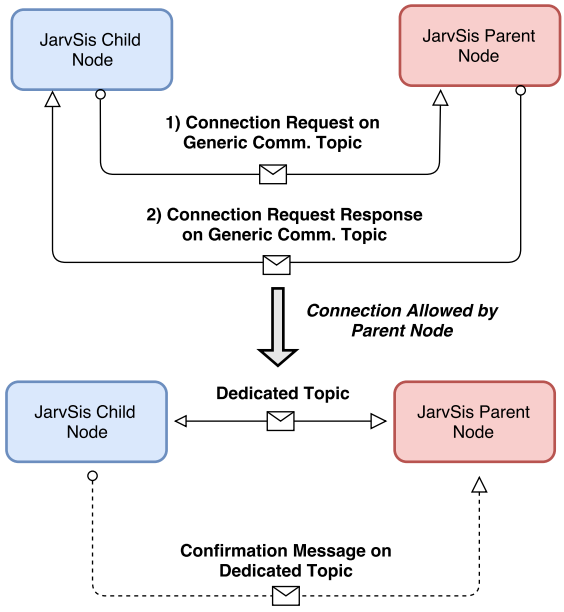

Figure 2.7: JarvSis Nodes Parent Child dedicated Topic

connected with him in the next lower hierarchical level, and with *(ii)* its own *parent* (it will receive directives), i.e. the JarvSis node directly connected with him in the next higher hierarchical level. In this way, if a node becomes unavailable, its parent node, that controls the nodes in the underlying layer, will be in charge of replacing the missing node with a new one, i.e. by redefining the tasks managed by the failed node to another node under its control. This will result in a certain level of resilience.

The current version of the JarvSis prototype is driven by an initial allocation of a fixed set of resources which are used to test basic mechanisms, as load balancing. In particular, tasks are initially allocated on resources by the directives included in the configuration files. The initial distribution of clusters may be also automated by a simple preliminary analysis of the user-defined cluster descriptions, given the task dependencies.

Load balancing is performed by moving clusters of tasks from a JarvSis node to another one, when needed. To this end, JarvSis lets the user to define specific policies (through the usage of the Adaptive Scheduler interface) to trigger the migration needed to balance the workload among nodes. The migration mechanisms provided and pre-defined within the JarvSis infrastructure are separated from the migration policies, which are defined and implemented by the user accordingly to the JarvSis Network configuration, the specific scenario and application.

Any user-defined migration policy must be composed by an algorithm that prevent the nodes overload, such that the execution of the single cluster is preserved in the overall network. Moreover, the separation between mechanisms and policies allow the definition of several and customized algorithms by the user; this capability for example could enable energy-aware computations [70] through the definition of a specific algorithm, capable to move the cluster/task from one node to

another one by analyzing the actual workloads and the related energy consumption.

Furthermore, the modular architecture of JarvSis supports the addition of pluggable modules that enable the exploitation of specific services for resource discovery and allocation, in order to allocate managing tasks in the Fog and in the public cloud. In particular, services for IoT resource discovery and allocation are useful to allocate functional tasks or to move them from a device to another one. Also in this case, the needed handles to communicate with these services can be encapsulated into specific modules to plug into JarvSis.

JarvSis performs the migration from a node to another one into three phases, as follows (the reader may refer to Figure 2.8):

- *Computation of candidate.* Basing on the data about the workload of the cluster, the user-defined policy selects the most suitable node—the *candidate*—to host the cluster.
- *Negotiation.* This phase is performed by the parent node and the selected candidate, to verify its availability to host the cluster that has to migrate. This availability depends on the policies specified in each single node and the compatibility of the node itself with certain types of tasks. If negotiation succeeds, the migration is attempted as specified below.
- *Migration.* This phase is actually executed if a negotiation has been successful. If, for some reasons, the migration fails, another node is selected by the first phase. There are two different types of migration, on the basis of the nature of the tasks to migrate. As tasks are stored in external applications, migration regards only the control agents (i.e. timers and threads, as discussed in Section 2.7) that monitor the task itself.

Scalability can be managed, in JarvSis, in different ways. In this case, a distinction must be made between managing and functional tasks. Indeed, JarvSis can provide the basic scalability mechanisms for computational resources needed by managing tasks (Cloud and Fog) because any (parent) node can scale-up/down resources for clusters of tasks by exploiting Cloud resource discovery and allocation services offered by the Cloud. This allows the application engineer to obtain the needed flexibility to add/remove JarvSis nodes at runtime. Scalability of IoT resources must be, instead, managed by the user, that defines not only the policies to move and/or replicate IoT tasks from a device to another one (e.g. collecting data from sensors) but also the needed modules to exploit the requested mechanisms (e.g. activating an additional number of sensors in a specific area).

2.7 Implementation

2.7.1 Core technology

The first version of JarvSis was developed as a Windows service in VB.NET [55], along with a Microsoft SQL server and a SignalR message bus used to perform communication among nodes.

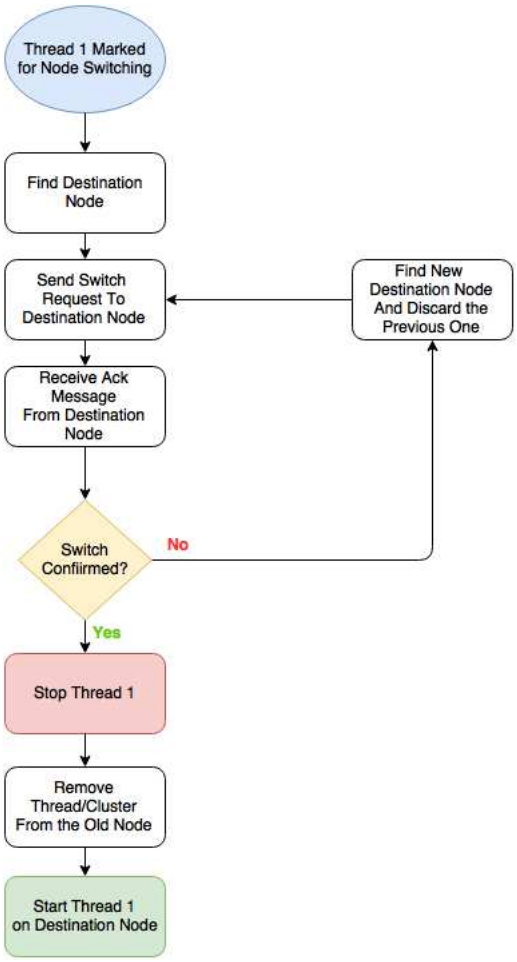

Figure 2.8: JarvSis Control Flow

To enhance the capability of JarvSis and allow also the usage within Linux based system/hardware the current version of JarvSis relies on a Java application developed , along with a Sqlite database and a IoT broker message bus, which is used to perform communication among nodes,external application and hardware.

The scheduling policy is implemented as a simple fair-share on the pools of threads allocated for the managing tasks of the user-defined clusters, and is also based on the task dependencies specified by the user, as explained below.

The tasks execution is managed by the Adaptive scheduler that is composed by two different sub-scheduler called "Parallel Scheduler" and "Sequential Scheduler". These sub-scheduler, accordingly to the task/cluster type execute the tasks through an implementation of the MultiThread controller.

The task starting type are the following:

- **Auto Start Timer**: the task will be executed after a pre-defined time interval in a dedicated thread. If the cluster is sequential the hierarchy of the task will be checked, and the task will

be executed only if the previous task is completed.

- **Activation Message**: the task will be executed if a specific activation message it's received by an external application or the Adaptive scheduler.

In case of a cluster composed by tasks activated by **Auto Start Timer** an array of n timers is defined, where n is computed on the basis of the task dependencies within the clusters: if all the tasks of the clusters are parallel (scheduling_type=0) then the number of timers will be equal to the number of tasks, in the opposite case one timer is enough; on the other hand, if the composition of the cluster is hybrid, the number of dedicated timers will be equal to the number of parallel tasks plus an additional timer for the tasks to be executed in sequence. Finally, when the cluster is composed of a number of tasks to be executed sequentially (with a **Auto Start Timer** starting methodology), a single timer is instantiated. Several callback functions are associated to different timers: these callbacks have indeed different behaviors, on the basis of the type of cluster (*parallel*, *sequential* or *hybrid*). For parallel clusters, the callback function executes the task without performing any check. Conversely, for a cluster of sequential tasks, the timer checks the execution of the current task, and starts the next one as soon as the current task has terminated its execution.

Listing 2.4: Instantiation of parallel tasks

```
public void initTasksLoop(){

                //init all the tasks that have task_activation_timer="act_timer"

                Iterator<Cluster> cluster_iterator=cluster_parallel.iterator();

                while(cluster_iterator.hasNext()){

                        ArrayList task_list=cluster_iterator.next().getTask_array();

                        Iterator<Task> task_iterator=task_list.iterator();
                        final Json_Builder jsb= new Json_Builder();
                        final configuration_settings cfg = new configuration_settings();

                        while(task_iterator.hasNext()){
                                final Task task_tmp=task_iterator.next();

                                //check if the task activation mode is act_timer (must be launched every
                                    X seconds)

                                if(task_tmp.getTask_starting_type().equals(jarvsis_internal_dictionary.
                                    task_activation_timer)){

                                        final Runnable task_executor = new Runnable() {
                                                public void run() {

                                                // check task status: 0 ready 1 running 2 completed
                                                if(task_tmp.getTask_status()==0 || task_tmp.
                                                    getTask_status()==2){
                                                        System.out.println("Sending Task exectution msg" +
                                                            task_tmp.getTask_name());
                                                        //String msg="start task" + task_tmp.
                                                            getExt_task_id();
                                                        //create activation msg
                                                        JSONObject act_msg= jsb.createTaskActivationMsg(
                                                            task_tmp);
                                                        //select pubblish topic /node/node_id/
                                                            ext_application_id
                                                        String pub_topic= jarvsis_internal_dictionary.
                                                            topic_prefix_extapp + "/" + cfg.getNodeID()+
                                                             "/"
                                                        + task_tmp.getExt_application_id();
                                                        System.out.println("topic: " + pub_topic);
                                                        System.out.println(act_msg.toString());
                                                        MqttMessage message = new MqttMessage(act_msg.
                                                            toString().getBytes());
                                                                mqtt_cl.publishOnTopic(pub_topic,
                                                                    message);
                                                        }
                                                }

                                        };

                                        //change timer to milliseconds
```

```
44                                          final ScheduledFuture<?> task_executorHandle = scheduler.
                                                scheduleAtFixedRate(task_executor,
45                                           task_tmp.getTask_timer(), task_tmp.getTask_timer(), TimeUnit.
                                                SECONDS);
46                                  }
47
48
49                              }
50
51
52          }
53  }
```

Listing 2.4 describe the previous described parallel tasks activation process, implemented within the "Parallel Scheduler" managed by the Adaptive Scheduler.

The task/cluster information are retrieved by the Sqlite database through a specific interface capable to extract all the information needed by each sub-scheduler

2.7.2 Task Management through MQTT protocol

JarvSis is able to controls,manage and execute tasks that are implemented in an external application through the usage of a MQTT protocol, a broker and a pre-defined messages structure. A task implemented in an external application can be for example a GO-To-Point function implemented in the software that controls a specific robot or a data-processing/sending routine defined in a sensor. In both cases the task will be invoked directly by JarvSis accordingly to a specific cluster/task structure. In the case of a task implemented within an external application, JarvSis's cluster/task structure will be only a representation of the task itself (that as previously described it's implemented within the robot operative system or control software/platform).

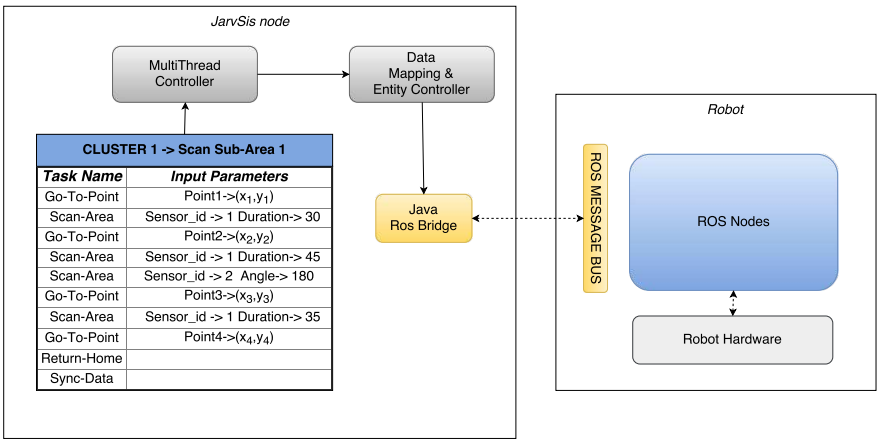

Figure 2.9: Native JarvSis node and ROS interaction

The robot cooperation and coordinated control is an interesting and tight topic, the usage of JarvSis in this environment is useful to describe its capability and the relationship between a JarvSis's task representation and its real implementation. Typically the robots platform are controlled and managed by a specific software application customized accordingly to the hardware of the robot and the tasks that it must execute. One of the most promising and used technology in this area is ROS (Robotic Operative System) [57], an open-source C++ based software used to manage and control several robots platform (UAVs,UGVs etc.).

The ROS architecture is based on several nodes that communicates through a shared communication bus called "ROS Message BUS". In particular each node can send and receive custom or standard message through a pre-defined interface able to connect the node to the message bus. The messages are organized and divided by different topic that can be defined also by the user accordingly to the nodes number and configuration. Within ROS eco-system each node can be used to implement a control algorithm,to integrate a specific component installed on the robot or processing the data produced by others nodes. In this section we consider ROS as an *external application* where the tasks are implemented and JarvSis as the *manager* of this tasks. This means that we want to control a robot task execution through a specific cluster/task structure defined within a JarvSis node.

Following the scenario previously described, if we consider a robot controlled by a set of ROS nodes, a JarvSis instance can send messages to the ROS message bus directly 2.9 on a specific topic through the Data Mapping and Entity Controller, while the status of the tasks is monitored by a subscription of a specific ROS topic, as the task status is published by ROS itself. In this case JarvSis it's directly integrated with ROS message bus through a specific Java interface customizable by the user.

This process is shown in Figure 2.9, that exemplifies the interaction pattern between JarvSis and ROS.

Furthermore if the robot and the related JarvSis node are connected to the same MQTT broker an alternative interaction mode can be used. In this last scenario the messages are received/send to the ROS message bus through a MQTT C++ interface directly implemented within ROS and customizable by the user 2.10.

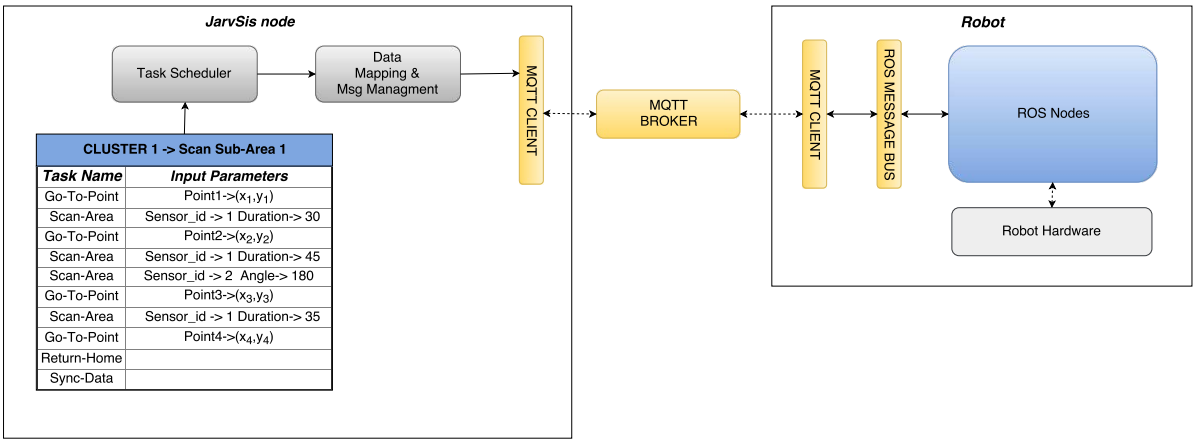

Figure 2.10: JarvSis node and ROS interaction through MQTT

This means that if we consider for example a robot controlled by a ROS software, it can be able to read and translate the messages received by a JarvSis node, through an MQTT/C++ client implemented within a ROS node (composed by several functions able to parse JarvSis's messages). In both cases, to exchange information between JarvSis and ROS, the communication interface

implemented within ROS must be customized accordingly to a specific messages structure and types defined within JarvSis. In the proposed approach JarvSis will provide a standard message structure and all the external application that want to communicate with JarvSis must implement a tiny interface to able to read and translate the messages received by a JarvSis node.

The JarvSis message structure used to invoke an external task execution is composed by three different messages:

- *Activation Message*: it is the message that JarvSis publishes on a specific topic to start a specific task along with the needed parameters. It is the equivalent of the invocation sent by means of the method "startTask" defined in Table 2.1.

Listing 2.5: Task Activation message

```
1
2  // the message will be published on the ext_application_id +
       ext_application_name topic
3  {
4          "sender_id":"19823123123909", // sender id at node level
5          "task_task_id": "1", //task id at node level
6          "task_cluster_id":"1", //cluster id at node level
7          "task_ext_id":"22", //ext task id given by ext application
8          "task_ext_application_id":"12", //ext application id
9          "msg_type":"start_msg",
10         "task_payload":{....} // task custom payload defined by the user or
               the external application
11 }
```

- *Monitoring Message*: it is the message that JarvSis publishes on a specific topic in order to monitor the status of the task. It is the equivalent of the invocation of the method "status" defined in Table 2.1. The monitoring messages can be a monitoring message request (Listing 2.6) – sent by JarvSis through the Adaptive Scheduler– or a monitoring message response (Listing 2.7), which is sent send by the external application in response to a monitoring request.

Listing 2.6: Task monitoring message Request

```
1
2  // the message will be published on the ext_application_id +
       ext_application_name topic
3  {
4          "sender_id":"19823123123909", // sender id at node level
5          "task_task_id": "1", //task id at node level
6          "task_cluster_id":"1", //cluster id at node level
7          "task_ext_id":"22", //ext task id given by ext application
8          "task_ext_application_id":"12", //ext application id
9          "msg_type":"monitoring_msg",
10         "task_payload":{....} // task custom payload defined by the user or
               the external application
11 }
```

Listing 2.7: Task monitoring message Response

```
1
2  // the message will be published on the ext_application_id +
       ext_application_name topic
3  {
```

```
        "sender_id":"19823123123909", // sender id at node level
        "task_task_id": "1", //task id at node level
        "task_cluster_id":"1", //cluster id at node level
        "task_ext_id":"22", //ext task id given by ext application
        "task_ext_application_id":"12", //ext application id
        "msg_type":"monitoring_msg_response",
        "task_status":"task_status_ready",
        "task_payload" :{....}
}
```

- *End Message*: it represents the message published by the external application with the results of the task execution. It is the equivalent of the invocation of the method "result" defined in Table 2.1. The task *End Message* is composed by a monitoring message response where the task status is set to *completed* (Listing 2.8).

Listing 2.8: Task End Message

```

// the message will be published on the ext_application_id +
    ext_application_name topic
{
        "sender_id":"19823123123909", // sender id at node level
        "task_task_id": "1", //task id at node level
        "task_cluster_id":"1", //cluster id at node level
        "task_ext_id":"22", //ext task id given by ext application
        "task_ext_application_id":"12", //ext application id
        "msg_type":"monitoring_msg_response",
        "task_status":"task_status_completed",
        "task_payload" :{....}
}
```

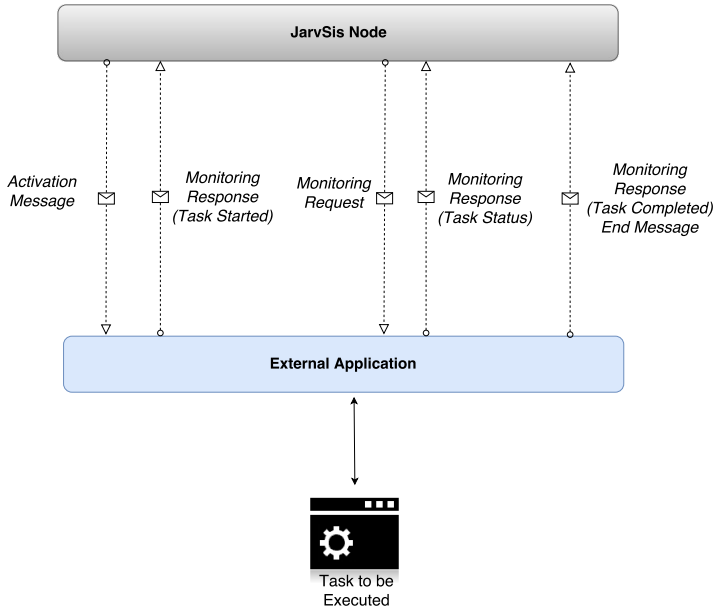

Figure 2.11: JarvSis node and External Task execution through MQTT

The previous messages type define the JarvSis interface to invoke and monitor external task implemented in external application. The overall flow composed by the previously defined message

are defined in Figure 2.11. The user that want use a JarvSis node to control and organize his tasks in a structured and defined way must implement a tiny data-mapping layer able to send and receive the messages defined by the JarvSis interface. This additional module can be implemented easily through a Json parser and a small set of rules to translate the received message and send the righ response.

2.7.3 JarvSis and Multi-Agent Simulations

This chapter has described the JarvSis architecture and technology the model of cluster/task organization, the implementation technology to demonstrate the capability of the approach used by JarvSis. The next chapter 3 describes a multi-agent simulator, called *AgentSimJs*, which is based on the JavaScript technology [23], will be described. In particular, AgentsimJs has been used to simulate the environment and several external application according to the case studies described in chapters 4 and 5.

3

Multi-agent simulation with AgentSimJs

Agent-based modeling and simulation (ABMS) is a powerful technique to model the collective behaviors of interacting autonomous entities [41, 32, 36], as proven by the many existing applications, e.g. supply chains, consumer markets [50], complex networks simulation [5], as well as the threat of bio-warfare and multi-robot simulations [66]. The foundations of ABMS have been studied in [41], and a large range of toolkits and developing methods can be found in the literature [40].

Furthermore we have assisted to the exponential growth of the technologies for building modern web applications – for example those for building SPAs (Single Page Applications) [49] – and a lot of applications, even traditional ones have been moving to the Web (e.g. Google Docs). It is well known that Web applications offer several advantages with respect to desktop oriented applications, for instance they are accessible from anywhere by a browser and they are easily customizable. Web applications easily enable interoperability and data sharing, which are key requirements for scientists that operate in the field of ABMS: for example, it is highly desirable that designers can share behavioral models and results, in order to e.g. compare the outcomes of different scenarios. JavaScript is one of the most used language for building modern web-based applications [23, 21] and a lot of web-related technologies are based on it.

AgentSimJs is a JavaScript 3D simulation framework of multi-agent systems capable to run in a web browser. This chapter discusses the design and the functionalities of AgentSimJs, which has been used to simulate the case studies described in Section 4 and 5.

AgentSimJs has been designed to stress the collaboration aspect, since, thanks to its modular architecture, code reuse can be properly addressed by sharing simulation code developed by different users. Two different components handle agent behavior and agent interaction, as well as the simulation of the physical environment and the agent-to-agent and agent-to-object collisions. In addition, the simulator offers the opportunity to distribute the simulation among several machines/thread through the component named Web-API integrator. AgentSimJs is capable to simulate a 3D physical environment, through the usage of *ThreeJs* library, where the agents can be defined y means of a custom 3D structure/texture and behaviors. This implies that each agent can interact with other agents and object also in a *physical* way. AgentSimJs is made by a number

of components able to define the environment and the agents, simulate the physical interaction among the objects/agents on the scene and define agents behaviors.

This chapter is structured as follows. Section 3.1 discusses background and related work, while section 3.2 contains a detailed description of the architecture of the simulator. Section 3.3 describes a detailed case study, as a proof-of-concept of the capabilities of the simulator, while section 3.4 discusses the integration of AgentSimJs with JarvSis.

3.1 Background and related work

As stated in the introduction of this chapter, the aim of agent-based modeling and simulation (ABMS) is to model the collective behaviors of interacting autonomous entities [19]. A wide range of applications can be mentioned, as supply chains, consumer markets [50], IoT applications [25, 1], complex networks simulation [5, 48, 42, 45, 11, 10, 44, 43], mobile agents systems [24, 26], threat of bio-warfare and multi-robot simulations [66, 53, 16, 27, 47].

One of the most popular toolkits for agent simulation is NetLogo [63], which is Java-based. Its development started in 1999 with the aim of modelling the evolution of hundreds or thousands of agents composing a complex system. An interesting feature is the authoring environment that allows researchers to share their own models. In particular, the Models Library, included with NetLogo, represents a collection of pre-built simulation models: biology, physics, computer science and so on. NetLogo gives the opportunity, to its users, to run into a "classroom participatory-simulation" tool called HubNet, which allows students to control an agent in a simulation through the network.

Another tool, named ComplexSim [46], is a C-based simulator capable to exploit SMP-aware systems to execute parallel simulations of complex networks on commodity multi-core hardware. Its architecture is based on two layers, the parallel simulation kernel and the complex network data and runtime. The simulator does not provide a graphical environment, as it is designed to be exploited for batch simulations. It provides a simple API which leads the programmer to implement his own data structures and to program the behaviors of entities in a very simple way.

Breve is a 3D simulation environment for simulation of decentralized systems and artificial life [39], which allows the designer to simulate continuous time and 3D space, and therefore is suited for a different class of simulations. It also includes an interpreted object-oriented language, an OpenGL display engine, as well as the support for the articulated body physical simulation and collision resolution with static and dynamic friction. Agent behavior is implemented in Python [29] or by another easy-to-use language named "steve". Breve is no longer maintained since 2009, but the simulation environment is still used, for this reason the author of Brave has partially restored the website[1]. Furthermore, a minimal JavaScript version of Brave is available online [2] and it allows the researcher to build simple simulations.

[1] http://www.spiderland.org
[2] http://artificial.com/breve.js/

Another 3D simulation tool is PALAIS [59], which is designed for prototyping, testing, visualization and evaluation of AI algorithms for games; it allows game designers to define and execute arbitrary, three-dimensional game scenes and behaviors. It also provides a scripting environment and a simple programming interface, simulation control and data visualization tools. In particular, as stated by the authors, the scripting interface is minimal and can be accessed via simple JavaScript. One of the interesting feature is that the PALAIS project can be easily shared, in order to collaborate with peers and build up showcases for algorithms and behaviors.

The project ARGoS [54] represents an open source, modular, multi-robot simulator to simulate real-time large heterogeneous swarms of robots. User can easily add custom features and allocate further computational resources where needed, moreover multiple physics engines can be used and assigned to different parts of the environment. The authors discussed some experimental tests proving that ARGoS is able to allocate and simulate about 10,000 simple wheeled robots 40% faster than real-time.

FLAME [9] is another agent-based modelling framework for high performance computing that allows designer to draw simulations even without parallel programming expertise. The authors tested successfully the efficiency of the simulator with a half a million of agents and 432 processors, and proved that a parallel efficiency of above 80% can be reached.

3.2 AgentSimJs Architecture

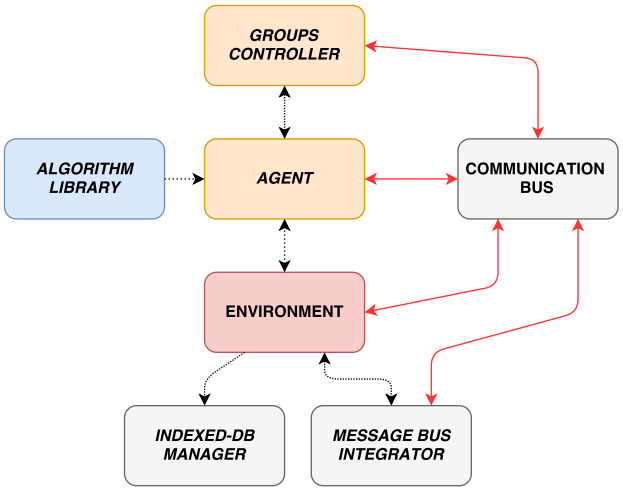

Figure 3.1: Architecture of AgentSimJs

AgentSimJs has a modular architecture made of several interacting components. They are represented in Figure 3.1, together with the primary connections, and first briefly presented here; they will be then described in more details in the following sections.

Agent represents the building block of the whole framework, as it contains the primitives used to program the agent behavior; such primitives include also the ones needed to perform communi-

cation among agents which is however handled and simulated by means of the **Communication Bus**.

The **Group Controller** implements mechanisms useful to realize group formation, while the **Environment** is used to handle physical interactions and (possible) agent-to-agent and agent-to-environment collisions.

The **Algorithms Library** contains a collection of primitives useful to design the building blocks of the agent behavior.

The **IndexedDB Manager** is used to store data related to the simulation in a temporary cache, that can be then exported at the end of the simulation.

Finally, the **Message-bus integrator** enables the deploying of several simulation on different machines by means of a shared environment.

In the following we describe the main characteristics of the components listed above.

3.2.1 Agent

Component `Agent` is a JavaScript class that exposes methods, representing specific functionalities, that can be also extended by the programmer according to her/his requirements. This class can be used to represent any software agent which can be also a physical entity (e.g. a robot) which lives in a certain physical environment. For this reason, class `Agent` provides a first set of functions related to the management of the agent motion, simulated by means of a spline interpolation of the path way-points (Figure 3.2 and Listing 3.1) and a standard collision detection algorithm.

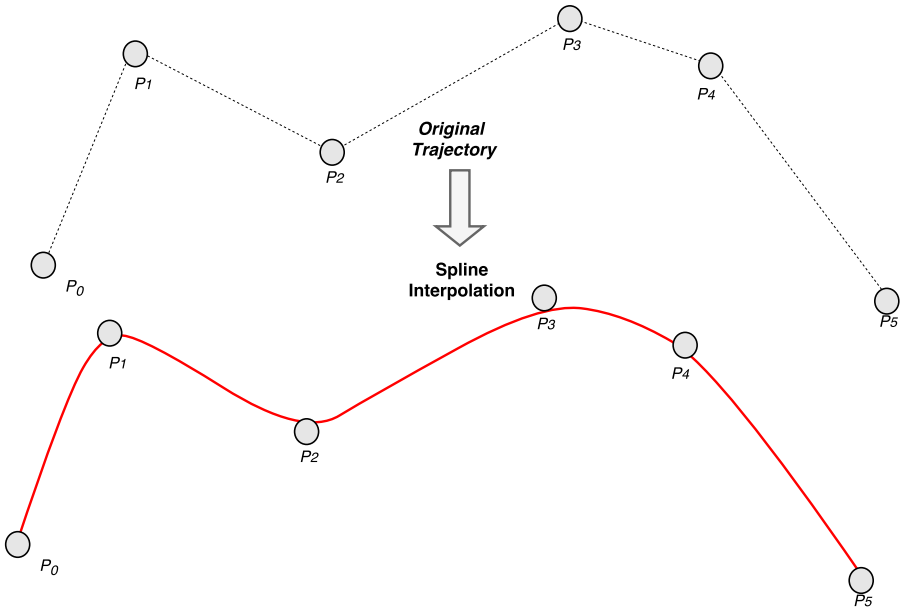

Figure 3.2: AgentSimJs trajectory cubic spline interpolation

These functions can be improved through the usage of a physics API (eg. ammo.js[3]) to take

[3]https://github.com/kripken/ammo.js/

into account aspects like agents shape, collision simulation, physics of the environment etc. In particular, each agent will be able to perform several predefined motions, e.g. linear, circular and parabolic.

A further set of functions provided by the class `Agent` is related to behavior programming. These are the basic functions related to communication and interactions with the environment. In this way, `Agent` class can be easily extended by the user that will take care only of the agent representation using the existing core functionality for the external interactions. To represent and handle the state of an agent, the `Agent` class includes the following properties (id, workers, motion status,position etc.) that can be retrieved through dedicated GET methods:

- **id**: represents the agent id assigned manually by the user during the agent definition.
- **agent_id**: represents the agent id assigned by ThreeJs to the 3D object associate to the agent.
- **movement_worker**: is the reference to the movement worker used by the agent while moving into the 3D scene.
- **spawn_point**: is the agent spawn point set by the user.
- **agent_object**: is the reference to the ThreeJs object that represent the agent on the 3D scene.
- **messagebus_worker**: is the reference to the message bus worker used by the agent to send/receive message.
- **indexdb**: is the reference to the indexedb worker used to store the agent positions data on the indexedb database.
- **agent_speed**: is the speed used by the agent to go across the user defined path.
- **collision_worker**:is the reference to the collision worker that estimates the collision and agents relative range.
- **disabled**: this parameter is related to the collision detection, if a collision with an object or agent is detected by the collision worker this parameter is set to TRUE.
- **motion_status**: this parameter represents the status of a motion task assigned to the agent.
- **agent_color**: this parameter can be used by the user to set the agent color (if the agent is represented on the 3D scene by a default 3D object).
- **use_custom_texture**: this parameter is set to TRUE if the user will set a custom texture/object to represent the agent on the 3D scene.

The agent position (x,y,z) is stored and updated by Threejs object (**agent_object**) and it can be retrieved by the user through the Threejs related primitives (eg. agent_object.position.x); for this reason the agent position is not explicitly declared as an agent parameter. The following methods are related to the interaction of the agent with the *Communication Bus*, thus allowing the user to simulate message exchange with other entities:

- **setMessageBus_Worker**: this method is used to set the message bus worker for the agent.
- **send_position_message**:this method is used to enable or disable the broadcast of the agent position through the message bus.
- **broadcast_message**: this method is used to allow the broadcasting of the received message (if a message is received by another agent it will be send back to the message bus).
- **setMessageListener**: this method is used to set the message listener callback function. Once a message is received by the agent it will be processed accordingly to the function indicated through this method.
- **processReceivedMessage**: is the standard/default function used to process the received message, it can be overridden by the user to implement a custom message processing.
- **setIndexdb**: this method is used to set the indexedb reference.
- **enableSavePosOnDB**: this method is used to enable or disable the agent's position saving on the indexedb database.
- **savePosition**:this method is used to save the agent's position on on the indexedb database.

Listing 3.1: AgentSimJs Agent motion along spline trajectory

```
1  function initSplineTraj(route,movement_worker,agent_speed,mov_type,
       motion_status,mov_worker_type){
2
3          var temp_agent_spline_init_msg=agent_spline_init_msg;
4          var path_point=path_point_data;
5          temp_agent_spline_init_msg.agent_vel= agent_speed;
6          temp_agent_spline_init_msg.spline_index=0;
7          temp_agent_spline_init_msg.agent_id=agent_obj.id;
8          temp_agent_spline_init_msg.ts=Date.now();
9          temp_agent_spline_init_msg.topicMsg="agent_pos_initspline_msg";
10         temp_agent_spline_init_msg.path_point_data=[];
11         temp_agent_spline_init_msg.agent_status="run";
12         temp_agent_spline_init_msg.motion_status="init";
13         temp_agent_spline_init_msg.motion_mode=mov_type;
14         temp_agent_spline_init_msg.mov_worker_type=mov_worker_type;
15         temp_agent_spline_init_msg.custom_texture=use_custom_texture;
16
17
18         for(var j=0; j<route.length;j++){
19
20                 var path_point= new Object();
```

```
21                  path_point.x=route[j].x;
22                  path_point.y=route[j].y;
23                  path_point.z=route[j].z;
24                  temp_agent_spline_init_msg.path_point_data.push(path_point);
25
26          }
27
28         /*************************************************************
29          The listener must be initialized only the first time to
30          avoid multiple agent representation and performace depletion!!
31
32         *************************************************************/
33
34         if(motion_status!="run"){
35
36           console.log("agent " + agent_obj.id + " listener initiated");
37           movement_worker.addEventListener('message', function(e) {
38
39           var received_msg=e.data;
40
41             if(received_msg[0].topicMsg=="motion_info"){
42                  if(received_msg[0].agent_id==agent_obj.id){
43
44                          agent_obj.position.x = received_msg[0].x;
45                          agent_obj.position.y = received_msg[0].y;
46                          agent_obj.position.z = received_msg[0].z;
47
48                          var axis=received_msg[0].axis;
49                          var radians=received_msg[0].radians;
50
51                          //rotate rotation axis
52                          agent_obj.quaternion.setFromAxisAngle( axis, radians
                                );
53
54                          if(!use_custom_texture){
55                          //agent compensation on roll axis if standard
                                texture agent is used
56                            agent_obj.rotateY(-(Math.PI - agent_obj.
                                  rotation.z));
57
58                          }
59
60                          if(message_bus_internal){
61
62                               send_position_message(agent_obj,agent_id);
63                          }
64
65                  }
66             }else{
67
68                   if(received_msg[0].topicMsg=="motion_end_msg"){
69                       if(received_msg[0].agent_id==agent_obj.id){
70
71                              console.log("Agent:" + agent_id + " has reached
                                   the final path's point");
72                              setMotionStatus(received_msg[0]);
73                       }
74                   }
```

```
75                }
76
77            },true);
78
79                this.motion_status="run";
80            }
81
82            //******Sending init/update path to movement worker**********//
83            movement_worker.postMessage(temp_agent_spline_init_msg);
84        }
```

The following methods instead are related to the *Environment*, i.e. the interaction between the agent and the environments, and the *Geometry*, as they enable the definition of the agent shape and the related reference system on the 3D scene.

- **buildAgentTexture**: this method is used to build the default agent 3D object/texture used to represent the agent on the 3D scene.
- **changeAgentTexture**:this method is used to set a specific 3D object/texture, defined by the user,to represent the agent on the 3D scene.
- **spawnAt setSpawnPoint spawn**:these methods are used to set a custom spawn point for the agent and insert the agent on the 3D scene.
- **setScene**:this method is used to set the 3D Threejs scene where the agent will be represented.
- **initSplineTraj initSplineTraj_render moveForward_Spline setMotionStatus**: these method are used to move the agent along the path set by the user (listing 3.1). The agent's path is approximated by a cubic spline through threejs functions that will be used by the movement worker to evaluate the agent position each time interval.
- **stop_agent**: this method is used to stop directly the agent motion.
- **findPointOnCircumference moveAroundPoint**:these methods are used to find a circumference around a specific point set by the user, build a path to reach the circumference from actual agent's position and finally move the agent algon the circumference.
- **plotTrajectory**:this method is used to plot the agent trajectory/path on the 3D scene.

Finally, the primitives related to the *Proximity sensor* are useful to make each agent aware of the environment.

A data structure is shared among agent instances, as it is used to store the history of the agent position or to set the path of the agents. Path planning is performed by setting high level strategies in the Group Controller, therefore, a path can be planned for a given group of agents. On the other hand, a fine tuning of the path can be performed by means of the API available for the motion of the single agent.

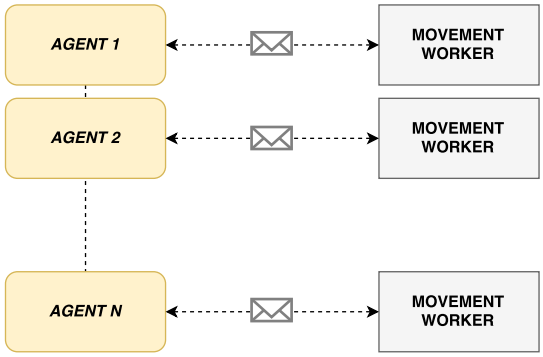

Figure 3.3: multiple movement workers

The methods available to get/set the current position of an agent can be exploited within a "movement worker". This situation is illustrated in Figure 3.3, while the different situation of a singleton worker is depicted in Figure 3.4. In the latter case, the singleton worker will have the responsibility to evaluate the position of each simulated agent at each simulation step by computing the next position of the agent by means of a (discretized) motion equation.

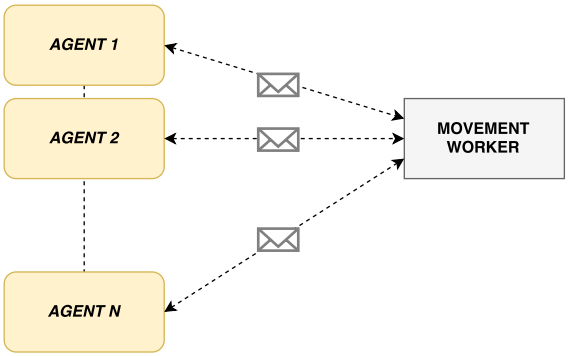

Figure 3.4: Singleton worker

In the centralized approach (singleton movement worker for n agents), the worker publishes n messages containing the updated agent position and the ID of each agent, that, once receives its own message, will update its own position with its own ID.

The AgentSimJs gent definition is simple and can be done with a small amount of code. In the Listing 3.2 an example related to the definition of two different agent is described. The first agent will move along a circumference (lines 39-41) while the second agent will follow a user defined path (lines 50-57/68). To define the agent 2 for example only a few lines of code are needed, line 10 for agent object definition, lines 60-62 for the agent 3D representation, lines 63-65 for workers settings and lines 68-68 to set the agent path and motion.

Listing 3.2: Agents Definition Example

```
1
2
3                 this.displayAxis = true;
4                 var agent,agent2;
```

```
5                 var movement_type='spline_motion';
6
7                 /********************** DECENTRALIZED MOVEMENT WORKER
                      EXAMPLE ***************************/
8                 agent= new Agent(1, 1, 2, this.displayAxis,new Worker('../
                      src/movement_worker.js'),
9                                                 movement_type,
                                                      collision_worker,"
                                                      decentralized");
10                agent2= new Agent(2, 0, 2, this.displayAxis,new Worker('../
                      src/movement_worker.js'),
11                                                movement_type,
                                                      collision_worker,"
                                                      decentralized");
12
13
14
15                //custom spawn point definition for Agent 1
16                this.spawn_x = 120;
17            this.spawn_y = 120;
18            this.spawn_z = 120;
19
20            //MSG bus and Indexed Db Manager Definition
21            var msgbus_worker=new Worker('../src/messageBus_worker.js');
22                var indexdb = new indexdb_manager();
23                indexdb.create_db();
24
25                // Agent 1 Definition
26            agent.agent_object=agent;
27            agent.setSpawnPoint([{x:this.spawn_x,y:this.spawn_y,z:this.
                      spawn_z}]);
28        agent.setScene(scene);
29            agent.setMessageBus_Worker(msgbus_worker);
30            //if set to false the position broadcasting through the message
                      buffer will be stopped
31            //agent.setPositionMsgMode(false);
32            agent.spawn();
33            agent.setIndexdb(indexdb);
34
35            //The agent will save its position history on the indexedDB
36            agent.enableSavePosOnDB(true)
37
38            //Agent 1 will reach the center of the circumference and the it
                      will move arout this point
39            agent.findPointOnCircumference(new THREE.Vector3(150, 100, -150)
                      );
40            agent.plotTrajectory(scene);
41            agent.moveAroundPoint();
42            agent.setMessageListener();
43
44            //custom spawn point definition for Agent 2
45                this.spawn_x = 220;
46            this.spawn_y = 320;
47            this.spawn_z = 320;
48
49
50                //trajectory definition for Agent 2
51                route_to_go=[];
```

```
52                  route_to_go.push(new THREE.Vector3(0, 0, 0));
53                  route_to_go.push(new THREE.Vector3(30, 50, 100));
54                  route_to_go.push(new THREE.Vector3(50, 50, 130));
55                  route_to_go.push(new THREE.Vector3(150, 50, 170));
56                  route_to_go.push(new THREE.Vector3(160, 50, 180));
57                  route_to_go.push(new THREE.Vector3(180, 50, 220));
58
59              //Agent 2 Definition
60              agent2.setSpawnPoint([{x:this.spawn_x,y:this.spawn_y,z:this.
                    spawn_z}]);
61          agent2.setScene(scene);
62              agent2.spawn();
63              agent2.setIndexdb(indexdb);
64              agent2.setMessageBus_Worker(msgbus_worker);
65              agent2.setMessageListener();
66
67              //Agent 2 will move along the defined path or route
68              agent2.initSplineTraj(route_to_go);
69                  agent2.plotTrajectory(scene);
```

3.2.2 Communication Bus

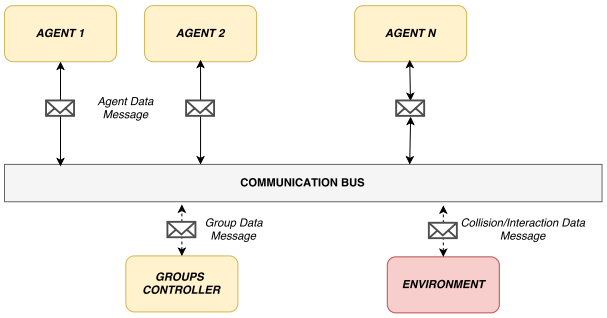

Figure 3.5: Communication bus

Through the Communication Bus, not only agents can exchange information, but all the components of the framework can exploit their functionalities. In particular, messages that can be exchanged by the communication bus can be classified into three main categories: *(i) Agent Data*, which is exchanged among agents to share information about, e.g. position, objects, environments, task etc.; *(ii) Group Data*, which represents data exchanged between the Group Controller and each member of the group; *(iii) Agent-Environment Collision/Interaction*, which represents information exchanged by the agents and the environment: such information are generally computed by the environment and sent to a specific agent. The several messages that can be handled by the Communication Bus are depicted in Figure 3.5.

The communication between the different components of AgentSimJs is asynchronous. In particular, the Communication Bus acts as a proxy for all the listeners. Each message is characterized by a specific topic, which helps the Communication Bus or a listener itself to recognize the real receiver. A number of predefined messages is used for the basic AgentSimJs functionalities (see Figure 3.3).

Listing 3.3: AgentSimJs Default Messages

```
1
2
3  ///*****DEFAULT-MESSAGE******///
4  var agent_pos_msg = [{
5              x: 0,
6              y: 0,
7              z: 0,
8              agent_id:0,
9              ts:"",
10             topicMsg:"agent_pos_msg"
11  }];
12
13 ///*****DEFAULT-MESSAGE******///
14  var agent_spline_init_msg = [{
15             agent_vel: 0,
16             spline_index: 0,
17             point_number: 0,
18             spline_len:0,
19             agent_id:0,
20             ts:"",
21             path_point_data:0,
22             spline:0,
23             agent_status:"run",
24             motion_mode:"foward",
25             motion_status:"init",
26             topicMsg:"agent_pos_initspline_msg",
27             mov_worker_type:"decentralized"
28  }];
29
30 ///*****DEFAULT-MESSAGE******///
31 var path_point_data=[{
32       x:0,
33       y:0,
34       z:0
35 }];
36
37 ///*****DEFAULT-MESSAGE******///
38 var agent_stop = [{
39             agent_id:0,
40             ts:"",
41             topicMsg:"stop_msg"
42  }];
43
44 ///*****DEFAULT-MESSAGE******///
45 var save_agent_pos_msg = [{
46             agent_pos:{ x: 0,
47             y: 0,
48             z: 0,
49             agent_id:0,
50             ts:"",
51             topicMsg:"agent_pos_msg"},
52             message_header:"save_agent_pos"
53
54  }];
55
56 ///*****DEFAULT-MESSAGE******///
57 var db_definition_msg = [{
```

```
58              database_name: "db_name",
59              message_header:"db_definition_data"
60  }];
61
62
63  //This message can be modified by specify the Hostname and the port of the
        MQTT Broker
64  var mqtt_broker_conn=[{
65                  topicMsg:"broker_managment",
66                  hostname:"noname",
67                  port:"",
68                  command:""
69  }];
70
71  ///*****DEFAULT-MESSAGE******///Verify if duplicated
72  //This message can be modified by specify the Hostname and the port of the
        MQTT Broker
73  var mqtt_broker_conn_start=[{
74                  topicMsg:"mqtt_conn_management",
75                  hostname:"noname",
76                  port:"",
77                  command:"open_connection" //others comm: close_connection
78  }];
79
80  ///*****DEFAULT-MESSAGE******///
81  var broadcast_message=[{
82          agent_broadcast:"0",
83          msg_payload:[]
84  }];
```

The user can customize messages and may use the Communication Bus capabilities to design a custom communication protocol among the agents. The difference between the message exchanged by the components and the agents is that the components of AgentSimJs use the messages already defined in the simulator, while the agents can use custom messages.

Figure 3.6 shows that, by means of the Message-Bus Integrator, several different actors – e.g. Group Controller and agents – can exchange simulation data from different hosts, without changing the interface used to exchange messages (i.e. the Communication Bus).

3.2.3 Overlay Network

By AgentSimJs, an overlay network of agents can be simulated. Messages can be very simple, i.e. they can contain the IDs of the sender and the receiver, as well as a custom payload. If an agent is outside the range of another agents, it will not receive any message from it. The mechanisms by which the overlay network is simulated in AgentSimJs can be described as follows (Figure 3.7):

- a specific component, the Collision Worker, will perform an evaluation of the agents positions that are within a given range (e.g. wi-fi range) and will send the information to each agents. The details of the agent selection algorithm are described later in subsection 3.2.4;
- the Communication Bus sends all the received messages to every agents connected to the Communication Bus (according to the overlay network structure and message propagation

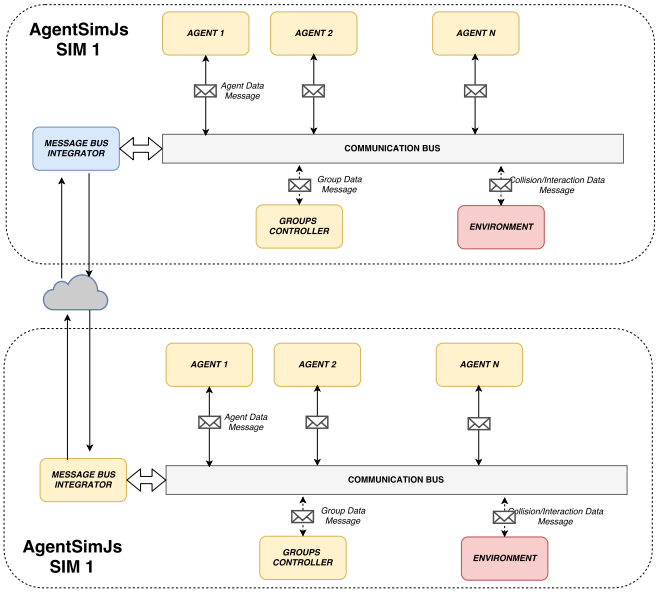

Figure 3.6: AgentSimJs Communication bus

algorithm);

- each agent will process the received message and if the Id of the sender is in the agents list that are in the selected range, the agent will store the message received in its internal message buffer. In particular:
  - it checks if its own Id is included in the Id list of the agents that have already read the message. If not, the agent must add its own Id into the list and eventually store the information and re-send the message to the near agents;
  - if the Id is already included in the list, the agent will discard the message.

3.2.4 Environment

The component called *Environment* is responsible of creating the graphical scene – which is defined by a set of 3D primitives that can be enhanced by the user – composed by a number of objects in a specific position and the computation of the possible agent-to-agent and agent-to-environment collisions. The collision are evaluated by a dedicate worker (the Collision Worker) that computes the mutual distances among objects and then compares these distances with a given threshold. A collision between the selected objects will occur if the distance is less than the threshold. The Collision Worker periodically calculates the relative position among the objects and the UAVs in the scene (lines 3-24 in Listing 3.4), and sends an array with all the objects within a certain range to every agents (lines 27-50 in Listing 3.4), in order to detect collisions.

Listing 3.4: Collision Worker Loop

```
1
2
```

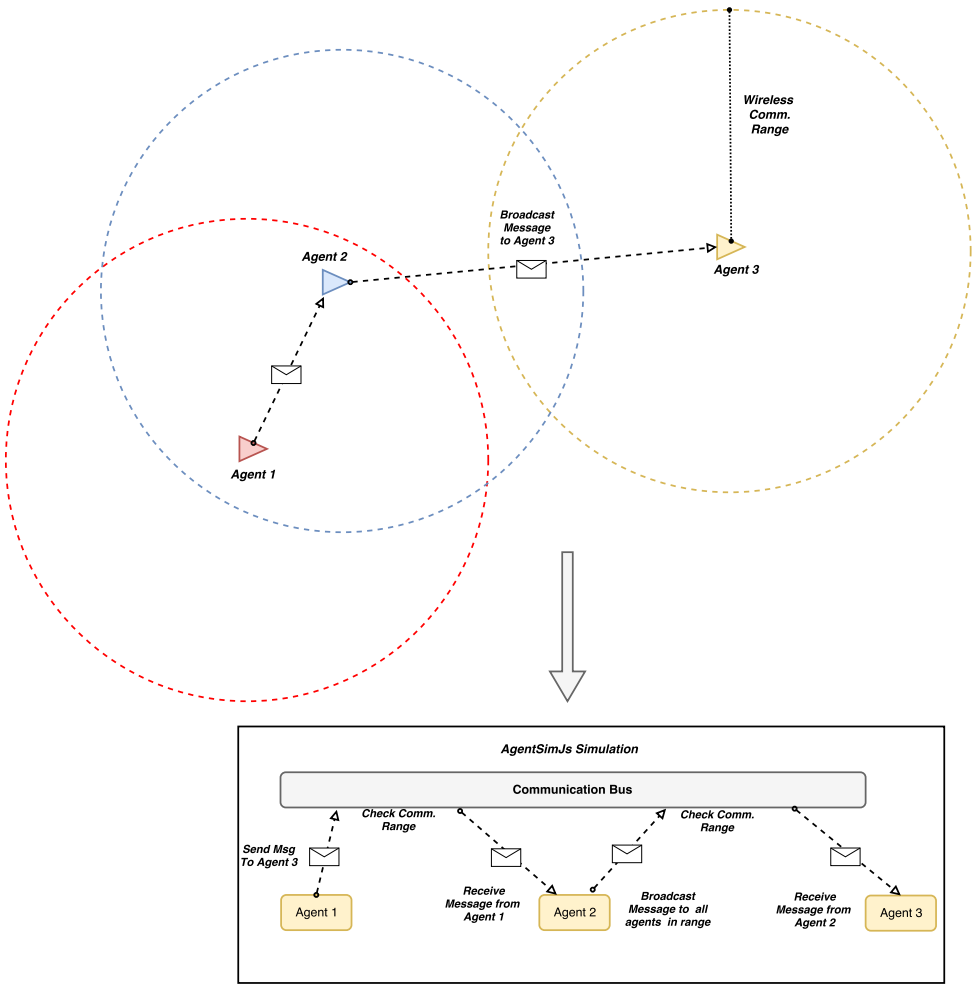

Figure 3.7: AgentSimJs Overlay Network through the Communication bus

```
this.checkForCollision = function(){

    var collision_message=collision_detection(this.object_ids)

    if(collision_message!=null &&collision_message.detected){
        console.log("COLLISION DETECTED object 1 ID:"+collision_message.
            object_id[0]);
        console.log("COLLISION DETECTED object 2 ID:"+collision_message.
            object_id[1]);
        if(collision_message.type[0]=='agent'){
            setDisabledInList(collision_message.object_id[0]);
        }
        if(collision_message.type[1]=='agent'){
            setDisabledInList(collision_message.object_id[1]);
        }

        self.postMessage(collision_message);
    }

    agent_relative_pos_eval(this.object_ids);

```

```
22      setTimeout(function(){checkForCollision()},loop_collision_time);
23
24  }
25
26
27  function collision_detection(object_ids){
28
29      for (var i = object_ids.length - 1; i >= 0; i--) {
30
31          //check the collisions/distance only for objects of agent type
32          if(object_ids[i].type=='agent'){
33              for (var k = object_ids.length - 1; k >= 0; k--) {
34                  if(object_ids[i].object_id!=object_ids[k].object_id){
35
36                      var distance=evalDistance(object_ids[i].x,object_ids[i].
                            y,object_ids[i].z,object_ids[k].x,object_ids[k].y,
                            object_ids[k].z);
37                      if(distance<collision_distance){
38                              var collision_message = {
39                              detected: true,
40                              object_id:[object_ids[i].object_id,object_ids[k]
                                    .object_id],
41                              type:[object_ids[i].type,object_ids[k].type]
42                          };
43                          console.log("Objects distance: "+ distance);
44                          return collision_message
45                      }
46                  }
47              }
48          }
49      }
50  }
```

The Collision Worker is also able to compute the range of the communication channel (eg. wi-fi) of each agent. Listing 3.5 shows a frame of JavaScript code (encapsulated within a Js function) to implement related functionalities. Function agent_relative_pos_eval is executed periodically by the Collision Worker. The goal is to select the set of agents within a certain range. To this end, an array is created by computing the relative distance between the last known position of each object (lines 3-19). If an object is placed within the chosen range (the array has a length greater than 0), then the Collision worker will send a message containing those information to the involved agents (lines 20-34).

Listing 3.5: Computations of the collision worker

```
1   function agent_relative_pos_eval(agent_objs){
2     for (var i = object_ids.length - 1; i >= 0; i--) {
3       //reset detected object array
4       this.agent_relativ_pos={};
5
6       //check if the agent is active
7       if(!object_ids[i].disabled){
8         for (var k = object_ids.length - 1; i >= 0; i--) {
9         //check only other agents
10        if(object_ids[i].object_id!=object_ids[k].object_id){
11          var distance= evalDistance(object_ids[i].x,object_ids[i].y,
```

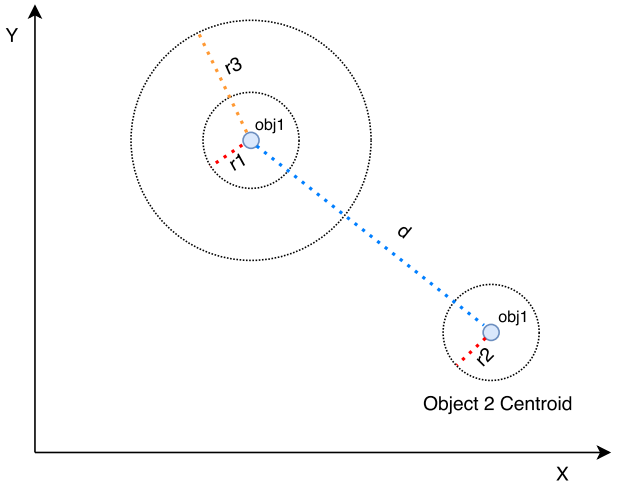

Figure 3.8: Collision detection

```
            object_ids[i].z,object_ids[k].x,object_ids[k].y,object_ids[k].z);
12          //check for wi-fi distance
13          if(distance<this.wifi_distance)
14            this.agent_relativ_pos.push(object_ids[k]);
15        }
16      }
17    }
18
19      //check if agent_relativ_pos is not empty
20      if(agent_relativ_pos.length>0){
21        var detection_message = {
22        detected: true,
23        agent_id:object_ids[i].object_id,
24        agents_relativ_pos:this.agent_relativ_pos
25        };
26
27        //return relative position to the agent
28        self.postMessage(detection_message);
29      }
30    }
31  }
```

Agents, once have received the information by the Collision Worker, start to process the messages they receive by the agents that are marked as "in range".

Collision detection is managed by a sphere-sphere technique: the collision worker stores the information about every object of the scene, which are updated once the object changes its position, while the worker evaluates the collision with a specific period that can be tuned accordingly to the simulation requirements.

As depicted in Figure 3.8, a collision will occur once the inequality $d < (r1 + r2)$ is satisfied. $R3$, instead represents the communication range of the agent itself. Once a collision is detected, the environment – which is the only component that is aware of the positions and status of every object on the scenes – sends a message to the involved agent(s), in order to set their state as

"off-line".

The simulation of the collision can be avoided by setting a given parameter. If a simulation is executed on several machines, then each of them will host an instance of the environment component. At runtime, if any environment has computed a collision, it will broadcast a message to the involved agents and the other environments through the Communication Bus. The other environments, once received the message, update the scene and the information about the involved agents and objects.

Since the collision detection technique is used to calculate the distances among objects, related distances are exploited to emulate message transmission among agents, and to emulate a proximity sensor to find all the objects near the selected agent.

3.2.5 Group Controller

The *Group Controller* allows the user to manage groups of agents: it exposes an interface to select the agents to form a group, to assign a GroupID to agents and groups, and to manage the groups during the simulation.

The Group Controller can send a broadcast message to any group in order to share specific information with the agents, as well as a heartbeat periodic message to monitor their status: an agent is off-line/not available if the environment detects any collision with another agent or an object; as a consequence, the environment will send a message to the interested agent, that will set its status to off-line. The Group Controller can manage the agents that are running in the same machine, i.e. it cannot exchange messages with the agents running in different machines. This choice is due to the network latency that may deteriorate the performance of the simulator.

3.2.6 Algorithms Library

The *Algorithms Library* provides an API useful to implement basic algorithms for the agents behavior. These methods can be invoked within any agent behavior. In this way the agents can share and reuse a certain knowledge (i.e. the algorithms), which will represent the "common logic" of the agents. The Algorithms Library can be enhanced by the users that will use the AgentSimJs framework.

3.2.7 IndexedDB Manager

The design of the framework includes a database to store a cache which is temporarily hosted inside the web-browser. The *IndexedDB Manager* is the resulting component which relies on the IndexedDB API [31]. Optionally information stored in the database can be sent to any remote database (e.g. to a Cloud DB through a Web-api) or exported in a standard format (e.g. CSV, XML, JSON) for future analysis.

Listing 3.6: IndexedDb Manager

```
1 this.create_db=function(){
2
```

```
3                 this.indexedDB_worker = new Worker("../src/indexdb_worker.js
                      ");
4
5                 var request = indexedDB.open(db_name,version);
6
7                 request.onerror = function(event) {
8                   ....
9                 };
10                request.onsuccess = function(event) {
11                  db = event.target.result;
12                  console.log("indexedDB correctly creadted/retrieved from
                        manager.");
13                };
14
15
16
17                request.onupgradeneeded = function(event) {
18                  db = event.target.result;
19                  position_store = db.createObjectStore("agent_positions", {
                        autoIncrement : true });
20                  position_store.createIndex("agent_id", "agent_id", {
                        unique: false });
21
22
23
24                  last_post_store = db.createObjectStore("obj_last_positions
                        ", { keyPath: "agent_id"});
25                  last_post_store.createIndex("agent_id", "agent_id", {
                        unique: true });
26
27
28                  position_store.transaction.oncomplete = function(event) {
29                        console.log("objectStore agent_positions,
                              obj_last_positions created");
30                  };
31                };
32
33
34                //send db parameters to worker
35                var temp_db_definition_msg=db_definition_msg;
36          temp_db_definition_msg.database_name=db_name;
37                temp_db_definition_msg.message_header="db_definition_data";
38                this.indexedDB_worker.postMessage(temp_db_definition_msg);
39
40
41          }
42
43
44  this.insertPosition=function(position){
45
46                //incapsulate message and send data to the worker
47                 var temp_save_agent_pos_msg= save_agent_pos_msg;
48                 temp_save_agent_pos_msg.agent_pos=position;
49                 temp_save_agent_pos_msg.message_header="save_agent_pos";
50
51                 //send data to database worker
52                 this.indexedDB_worker.postMessage(temp_save_agent_pos_msg);
53
```

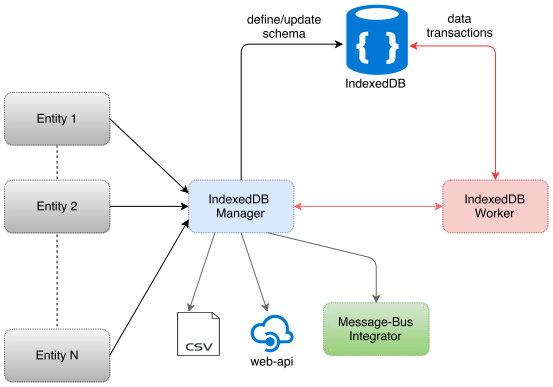

Figure 3.9: Indexed DB

```
54
55 }
```

The IndexedDB Manager is coupled with a IndexedDB Worker that takes care of storing the selected information (e.g. agents position) in a dedicated thread separated from the UI/Simulation thread (Figure 3.9).

The IndexedDB Manager is responsible of supporting the simulator database and its interface with the other components. Through the IndexedDB Manager the user can create a custom database schema by adding new documents and related methods (lines 3-24 in Listing 3.6). The logic of the database is defined within the IndexedDB Worker, where the methods related to data manipulation (insert/update/delete) are stored (lines 26-68 in Listing 3.7).

Listing 3.7: IndexedDb Worker

```
1
2
3  self.addEventListener('message', function(e) {
4
5                  var received_msg=e.data;
6                  //check msg header
7                  if (received_msg.message_header!==undefined){
8
9                                          if (received_msg.message_header=="
                                               db_definition_data"){
10                                             //retrieve indexeddb
11                                             console.log("starting
                                                   db_init on worker");
12                                             db_name=received_msg.
                                                   database_name;
13                                             retrieve_db();
14                                         }
15
16                                         if ((received_msg.message_header=="
                                               save_agent_pos") && (db_init)){
17                                             //save position on indexeddb
18                                             insertPosition(received_msg.
                                                   agent_pos);
19                                         }
20                 }
```

```
21
22
23  });
24
25
26  function insertPosition(position){
27
28          if(db_init){
29
30                  var temp_pos=[{x: position.x,
31              y: position.y,
32              z: position.z,
33              agent_id:position.agent_id,
34              ts:position.ts,
35              topicMsg:position.topicMsg}];
36
37                  var transaction =db.transaction(["agent_positions"], "
                        readwrite");
38
39                  var objectStore = transaction.objectStore("agent_positions")
                        ;
40                  transaction = objectStore.add(temp_pos);
41
42                  transaction.oncomplete = function(event) {
43                    //console.log("transaction initialized");
44                  };
45
46                  transaction.onerror = function(event) {
47                    console.log("error while saving agent position");
48                  };
49
50                  transaction.onsuccess = function(event) {
51                      // event.target.result == customerData[i].ssn;
52                      console.log("position added!");
53                  };
54          }
55
56  }
57
58
59  function updateLastPos(position){
60
61          var objectStore = db.transaction(["obj_last_positions"], "readwrite"
                ).objectStore("obj_last_positions");
62          //var index = objectStore.index("agent_id");
63
64          var transaction = objectStore.get(position.agent_id);
65          transaction.onerror = function(event) {
66                  //if not present insert the position on db!on error???
67
68          };
69
70          transaction.onsuccess = function(event) {
71            // Get the old value that we want to update
72             console.log(event.result);
73
74          if(event.result==undefined){
75                  objectStore.add(position);
```

```
76                  console.log("position added!");
77          }else{
78            // Put this updated object back into the database.
79            var requestUpdate = objectStore.put(position);
80             requestUpdate.onerror = function(event) {
81               // Do something with the error
82             };
83             requestUpdate.onsuccess = function(event) {
84               // Success - the data is updated!
85             };
86
87             console.log("position updated!");
88            }
89
90          };
91
92  }
```

3.2.8 Message-bus Integrator & MQTT

The aim of the *Message-bus Integrator* (Figure 3.10) is to add the capabilities to distribute a single simulation into several machines through a web-browser. To enhance the external connection capability of AgentSimJs, the Message-Bus Integrator is coupled with an MQTT Message Bus manager that is able to propagate the message received from the AgentSimJs Communication Bus to a MQTT Message bus (Figure 3.10).

The propagation of agents position through the MQTT Manager is implemented by default in AgentSimJs as described in 3.8. The message received by the Communication Bus are checked to verify if are related to agent's position and then published to a specific broker and topic. This capability can be enhanced by the user through a customization of the MQTT Manager by the user. However to integrate several AgentSimJs simulation scenario that run on different machines only the information about the Agent positions are required and this feature is implemented within AgentSimJs as a basic functionality.

This capability allows the designer to connect any device to the simulator, for instance it can be exploited to connect devices equipped with ROS [57] or can be used to represent the status and the position of a real agent. Indeed, through the MQTT message bus each robot/UAVs can send its information to the simulator and the user will be able to process received information with AgentSimJs easily.

The MQTT topic management is performed by the Communication Bus and the Agent itself but if needed the data-topic/mapping can be also performed with the MQTT manager by customizing the interface with the MQTT Broker.

Listing 3.8: MQTT and Message Bus Integration

```
1
2
3   this.setMsgBus_worker=function(msgbus_wr){
4
5       this.message_bus_worker=msgbus_wr;
```

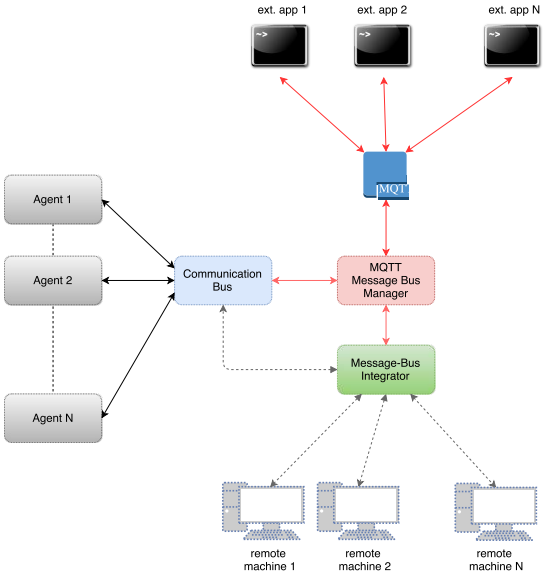

Figure 3.10: MQTT and AgentSimJs

```
6
7     this.message_bus_worker.addEventListener('message', function(e) {
8
9         if((e.data.topicMsg=="broker_managment")&&(e.data.command=="
              connect_toBroker")){
10            console.log("connect to broker");
11            connectToMqttBroker();
12        }else{
13
14            if(pubblish_pos_msg){
15              if(counter<message_package){
16
17                  //parsing data to pubblish on MQTT msg bus
18                  //check if the message is a position msg
19                  var obj = new Object();
20                  obj.x = e.data.x;
21                  obj.y  = e.data.y;
22                  obj.z = e.data.z;
23                  obj.agent_id=e.data.agent_id;
24                  obj.ts=e.data.ts;
25                  obj.topicMsg=e.data.topicMsg;
26
27                message_buffer.push(obj);
28                obj=undefined;
29                counter++;
30              }else{
31                  //pubblish agent position data on a MQTT Broker and Topic
32                  pubblish(default_topic, JSON.stringify(message_buffer), 0,
                        false) ;
33                  counter=0;
34                  message_buffer=[];
35              }
36            }
37
38        }
```

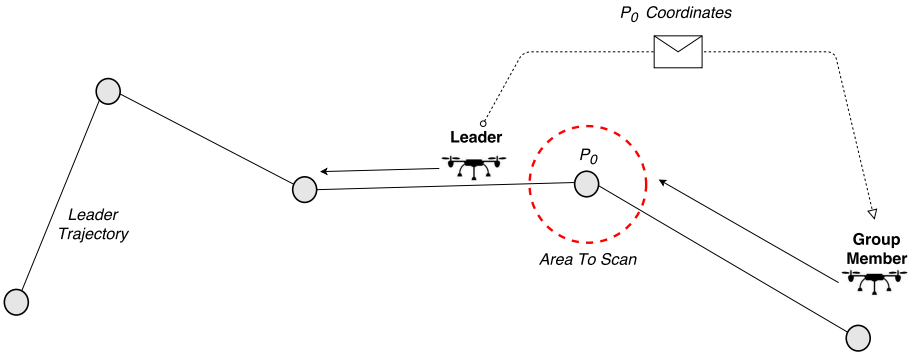

Figure 3.11: Leader and Target Area inspection

```
39
40
41      });
42  }
```

3.3 Case Study:Leader Following

In order to show the *usability* of AgentSimJs, it has been used to simulate a scenario on which two UAVs [35, 17] are employed to execute an aerial inspection in a certain area of several photo-voltaic plants. The details of the mission, as well as its design and implementation, are provided in the following.

The mission planning is quite simple: we assume that a flock of two UAVs is instructed to perform an inspection through a pre-defined path that allows it to reach any given plant. The flock is guided by a *leader*, which plans the path and sends instructions to the other UAVs. In particular the leader communicates to the second UAV to execute a specific inspection on a target area. Based on the received data the second UAV then will perform the inspection on the targert area.

This distributed collaborative process among the two agents of the is sketched in Figure 3.11.

Listing 3.9: Setting a simulation

```
1  //init collision worker
2  var collision_worker =
3    new Worker ('collision_worker.js');
4
5  var uav_number=6;
6  var plant_number=10;
7  var leader_id=4;
8
9  //define message bus worker and
10 //link the agents to the comm bus
11 var msgbus_worker =
12   new Worker('messageBus_worker.js');
13 var collision_worker =
14   new Worker('collision_worker.js');
15 //define group controller
```

```
16 var group_controller= new GroupController();
17
18 //UAVs
19 var uavs=[], agent;
20 for (var i=1;i<=uav_number;i++){
21   agent = new Agent(i,..,
22     new Worker('movement_worker.js'),
23     ...,collision_worker);
24
25   agent.setMessageBus_Worker(msgbus_worker);
26
27   //set the leader
28   if(i==leader_id){
29     agent.setAsLeader();
30     agent.setGroupController(group_controller);
31   }
32   uavs.push(agent);
33   //pass uavs to the group controller
34   group_controller.addAgent(agent);
35 }
36
37
38
39 //Plants
40 var plants=[];
41 for (var i=1;i<=plant_number;i++){
42   var agent =
43     new Agent(i,..,
44       ...,collision_worker);
45   agent.setMessageBus_Worker(msgbus_worker);
46   plants.push(agent);
47 }
48
49 //define leader trajectory
50 route_to_go=[];
51 route_to_go.push(new THREE.Vector3(0, 0, 0));
52 route_to_go.push(new THREE.Vector3(30, 50, 100));
53 route_to_go.push(new THREE.Vector3(50, 50, 130));
54 route_to_go.push(new THREE.Vector3(150, 50, 170));
55 route_to_go.push(new THREE.Vector3(160, 50, 180));
56 route_to_go.push(new THREE.Vector3(180, 50, 220));
57 //set the trajectory into the leader and start the Leader Motion
58 agent[leader_id].initSplineTraj(route_to_go);
```

UAVs are simulated by means of different agents, enabled to exchange messages, as follows. The Flock Leader is an instance of the class Agent that sends commands to the UAVs by means of the Message Bus. The second UAVs is an agent that follow the Leader in its path and accomplish the tasks assigned by the Leader.

An important aspect related to the implementation is the communication among agents: the Communication bus allow the developer to operate by customizing the content of the messages and the agent behaviors.

From a practical point of view, in order to define the scenario (agents, plants, communication

bus, and so on) the user can extend the basic AgentSimJs scene [4], which is a built-in module. Then, the simulation can be designed by implementing a JS script, by which the user can define the agents geometry, link each agent with a worker, create a Communication Bus and define the path.

A message `agent_pos_msg` (lines 25-32) is broadcasted by each UAVs to inform the peers about its own position and status, while a message `agent_spline_init_msg` (lines 36-49) is sent to the movement worker by the UAVs and is used to set the UAV speed (agent_vel), the path that must be followed and the motion type (*foward:* follow the path and stop the agent on last path point to perform the inspection). The Leader send a message to every agents when its position is updated by the movement worker. The message is then processed by the communication bus and sent to the listening agents. In our case there is only a second agent that receive the messages; it will skip a pre-defined amount of received message to simulate the definition of a target point in the Leader path. With this approach we can leverage on the AgentSimJs predefined functions and capability to reproduce the scenario previously described, but the user can implement a custom message structure and target point definition algorithm to send only a single message to the second agent.

Listing 3.10: Following Agent Message Processing

```
1
2
3  // to simulate the target point in the Leader Path
4  // a pre-defined amount of message will be skipped
5  var counter=20;
6
7  agent2.processReceivedMessage= function(msg){
8
9   //skip the message produced by the agent itself
10  //and process the message generated by the leader
11     if(msg.agent_id==1){
12             if((counter==20) && (agent2.status!="busy")){
13
14                 var point_to_go=[];
15
16                         //create the new path composed by the actual agent
                               point and the target position
17                 point_to_go.push(new THREE.Vector3(agent2.agent_object.
                       position.x,
18                                 agent2.agent_object.position.y,agent2.
                                       agent_object.position.z ));
19                 point_to_go.push(new THREE.Vector3(msg.x, msg.y, msg.z));
20
21                 //set the new path and reach the target position
22                 agent2.initSplineTraj(point_to_go);
23                     agent2.plotTrajectory(scene);
24                     counter=0;
25
26                     //set the agent status as "busy" until the task
                           completion
27                     agent2.status="busy";
```

[4] We will provide the detailed documentation along with the first release of AgentSimJs.

```
28
29                      //agent 2 will broadcast the message received by agent 1
30                       agent2.broadcast_message(msg);
31
32              }else{
33                  counter=counter+1;
34              }
35
36          }
37  }
```

The leader can also implement a selection criteria methods (in case of multiple UAVs) and select the most suitable agent to accomplish a particular task accordingly to the simulation scenario. This capability is allowed by the agnostic structure of AgentSimJs message bus that permits to customize the messages exchanged by the agents. In this way the user can extend the basic AgentSimJs messages in order to customize their content in order to develop and test agent behaviors and distributed algorithms.

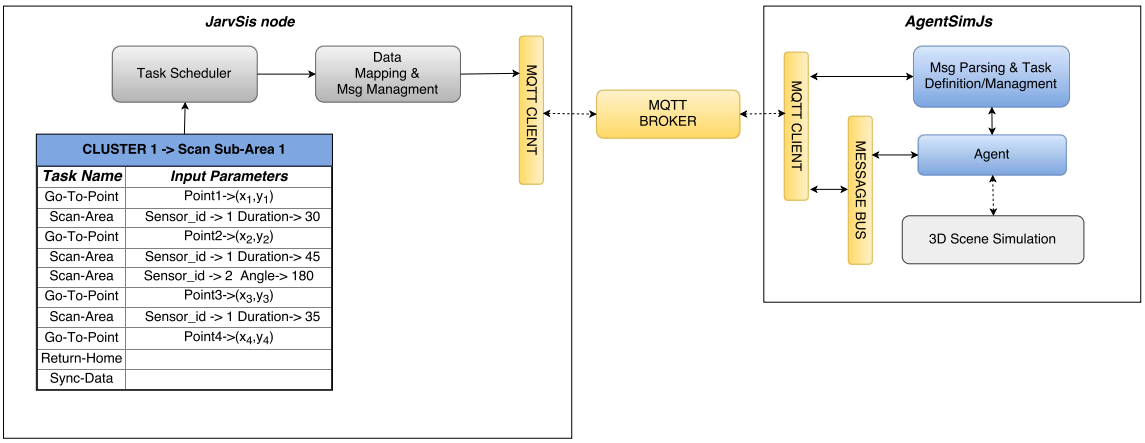

Figure 3.12: JarvSis Node - AgentSim interaction

3.4 Integration with JarvSis

In order to perform a simulation of the use-cases described in the following chapters we will use AgentSimJs as the simulator of the agents involved in different scenarios. This implies that a specific task must be defined in a JarvSis node and implemented within AgentSimJs (if the task is related to a specific functionality of an external platform it must be implemented outside JarvSis).

To execute then the task the two systems (JarvSis and AgentSimJs) must be able to communicate and exchange information about the task execution and status. In particular, a JarvSis node must be able to send/receive MQTT pre-defined messages from the agents defined within AgentSimJs. Therefore, in order to establish an MQTT channel an external MQTT broker is used and a specific simulation topic is defined (Figure 3.12). This connection can be easily established in AgentSimJs through the usage of the MQTT manager component, while JarvSis has a native

MQTT client used also to establish a communication channel with other nodes within a JarvSis network.

Any message sent by a JarvSis node, once received by AgentSimJs, must be processed according to the standard provided by JarvSis and the task implemented within AgentSimJs. For this reason a specific component must be defined in the AgentSimJs simulated scenario capable to parse the received message, execute the selected task through an agent and then send back to the JarvSis node the monitoring/ending message.

The connection mode previously described will be used in both the use-cases represented in the following chapter. Using AgentSimJs ensure a fast implementation of the scenario and the agents behaviours (task), furthermore all the algorithms involved can be implemented using the AgentSimJs components through JavaScript technology. This will allow us to focus on the cluster/-tasks/algorithms description and design without invest a huge effort in the simulation development side.

4

Case Study: JarvSis as an IoT platform System Integrator

The number of different IoT devices used nowadays is increasing rapidly, as well as the several IoT platforms able to gather data and send command to each device. Moreover, integration of such devices must be properly addressed by suitable software layers designed in accordance to specific integration requirements. To this end, JarvSis – as it has been discussed in Chapter 2 – is capable to support – by design – the necessity of integrating heterogeneous IoT devices. First of all, by means of the adoption of the MQTT protocol along with a standard format for the messages, enables JarvSis to propagate a payload along networks of IoT devices (e.g. sensors) managed by different software/platform and gateway. Secondly, the model of task organization and aggregation described in Chapter 2 represents a powerful tool to aggregate and control IoT tasks. Moreover, the lightweight implementation of JarvSis enables its execution in very small Linux-based devices (e.g. a raspberry pi-zero [30]), which means that a local and off-line integration among the IoT device can be implemented (e.g. within a single tiny gateway).

In the following a case study on the integration of different IoT device management through JarvSis is illustrated, along with the results of a number of simulations.

4.1 Simulation Environment and Context

Monitoring of critical areas is one of the most interesting and promising application of IoT technology and IoT based wireless sensors. In this case study we take in consideration a specific area where some wireless sensors are deployed to monitor the status of a terrain. The sensors are used to measure several terrain's parameters like humidity, vibration, altitude, composition etc. Typically, in this context, the sensors can have different connection capabilities (Figure 4.1):

- the sensors may form a mesh network and communicate with a local gateway which is responsible to gather data and, thereafter, send the information to a remote IoT platform through a broker.
- the sensors are capable to connect to a local broker by using the MQTT protocol, and the gateway will send the information to a remote application through a dedicated WEB-API.

In the scenario described here, a team (or flock) of drones is used to deploy the sensors in the target area, and a base station is deployed in the center of the target area, in order to gather data collected by the sensors.

The deployment of the sensors is driven by the following requirements:

- if the sensors form a mesh network, they must be deployed at a reciprocal distance which is less than the maximum relevant radio transmission range, such that the network will maintain its connectivity;
- if the sensors are able to connect to the gateway, they must be deployed within the maximum transmission range of the gateway itself.

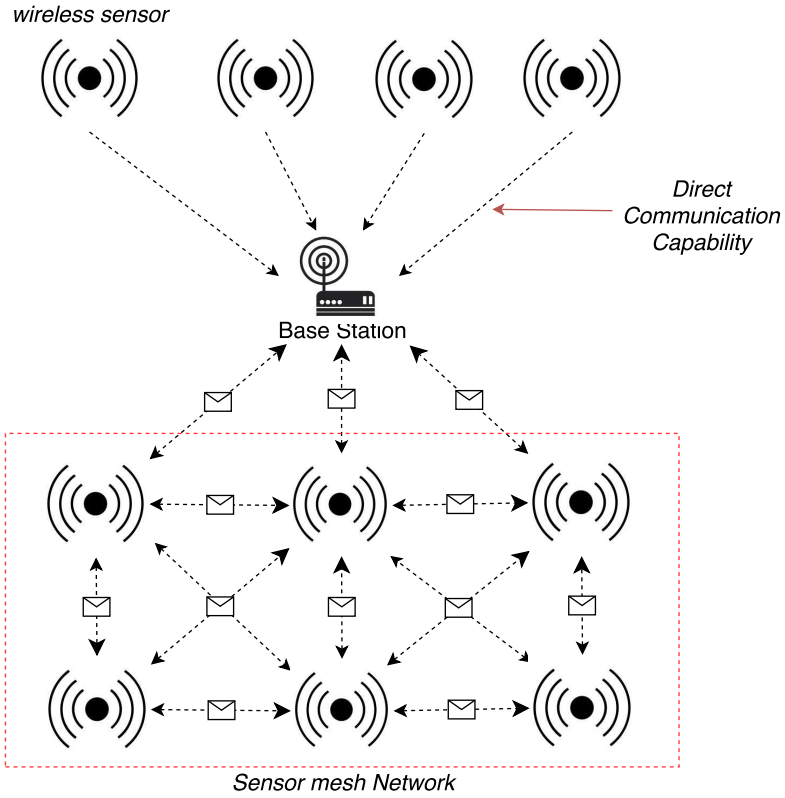

Figure 4.1: Deposition rate

The base station hosts a broker MQTT that is used by both the JarvSis node and the sensors to communicate. Furthermore, the base station must be capable to host a proprietary software interface to gather data coming from the sensors that use a proprietary wireless communication protocol (e.g. Zigbee) [71]. In this particular case the received data must be sent to JarVis trough a local MQTT client within the sensor interface software or by creating an ad hoc interface within the JarvSis node. The base station, once received the information from the sensors, will execute two different tasks accordingly to the sensors type (Figure 4.2):

1. In case of sensors capable to connect to the base station through an MQTT protocol the data received is grouped in a pre-defined array within a specific JSON and sent to a remote server through a RestFul Web-API.

2. In case the sensors form a mesh network and sent data to the base station through a proprietary protocol, received data is transformed in a JSON format, encapsulated in a MQTT message and sent to a pre-defined remote Broker.

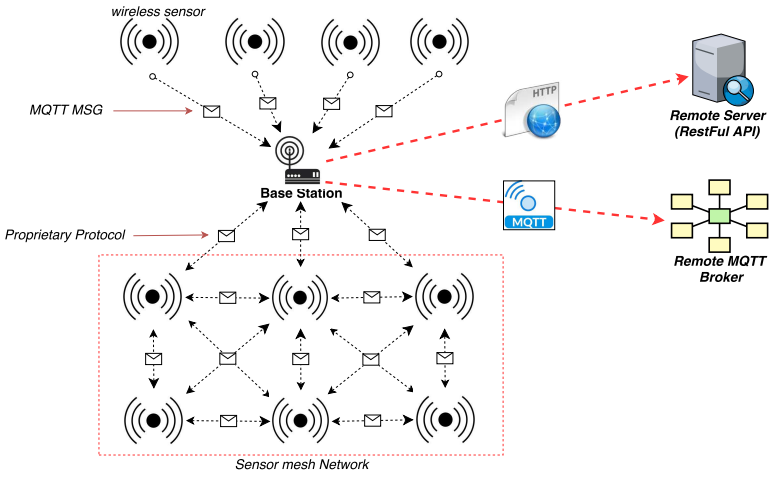

Figure 4.2: Base Station double remote communication Capability

The challenge behind the scenario illustrated above is represented by the fact that, very often there is the real need of using multiple sensors from different vendors, that cannot be replaced with different sensors that use a uniform protocol. In this complex scenario the sensors that use MQTT protocol may group gathered data and perform a local pre-processing before the data is sent to a remote application.

Instead the sensors that are not able to use a MQTT protocol will need to be integrated into an external application that use an MQTT protocol to gather the data from the field directly on the local gateway without using a cloud App-To-App layer (that can introduce additional latency and complexity to the integration of heterogeneous application process).

Typically in case of sensors that use a proprietary protocol to send data, a local gateway is used to collect this data and send them to a remote application hosted on cloud. Without direct MQTT local connection capability this data can be integrated and shared with external software application through an Application-To-Application layer composed by a RestFul Web Api exposed by the remote application responsible to collect the data from the gateways on field. In this configuration the external application must be able to consume this specific web-api and implements a data-mapping layer to translate the information in the most suitable format for its internal processing. This approach introduce a complex development scenario (a specific data mapping

layer must be implemented for each different platform that must be integrated) and implies an higher latency because the data must reach the cloud, processed by a specific application and then exposed by a Restful Web-API. Moreover to retrieve the data the web-api must be consumed through an HTTP/HTTPS request that implies additional latency on the previous one.

In case of sensors organized as mesh networks, we assume that all sensors are identical, having a maximum transmission range ρ (in meters), which represents the upper bound on the distance between sensors. Moreover, any sensor failing or exhausting its energy will start transmitting at a lower power, and it will not reach its neighbors, in this case the network may divide into two or more networks. This would result in the inability to collect data sensed in the partitioned zone. To avoid such problems, a classical rule-of-thumb is to spread sensors at a mutual distances significantly lower than ρ; the lower the (average) sensor distance, the higher the safety achieved with respect to adverse events such as the above-mentioned ones.

Tuning sensor distance will result into different *spatial densities*, the more critical the area to monitor, the greater the sensor density. In particular, to ensure a long-lasting data sensing, for a single area we use the following approach: *(i)* place a base station in the "center" (or *centroid*) of the hot zone; and *(ii)* distribute sensors at a spatial density which gets higher as we approach the center of the area. The above approach will ensure that the lower the distance of a sensor from the center of the area, the greater the number of its neighbors, the lower the average distance from them, and the lower the energy needed for sensor operation.

More formally, the scenario described before can be modeled as follows. We consider a single area with a circular shape; should the real area have a different shape, we can always find the minimum circle which contains the area itself. The circle is centered at a position $C = (x_c, y_c)$ (geographic coordinates), where the base station is placed (Figure 4.3), it has a radius R (in meters) large enough to include the selected area.

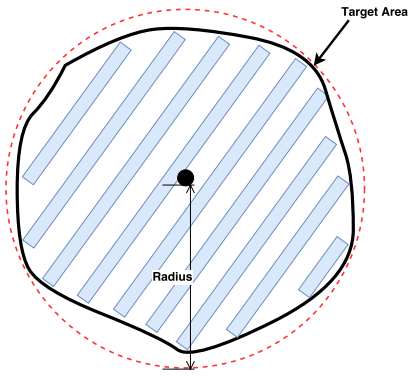

Figure 4.3: Circular Area Decomposition

In case of multiple areas that could overlap, since each circle has its own base station and each network has its own WSN-ID, we may safely assume such areas do not interfere with each other. For this reason, we limit our analysis to a single circle: the algorithm can then be simply replicated

if more than one circular area has to be covered.

In this scenario, to ensure that the sensors which are able to connect to the base station are placed in a proper way, we assume that the radius of the area is lower than their transmission range.

Within each circular area, sensors are to be deployed in accordance with a specific spatial density law $\delta(P)$, with P a point in the plane area; the density law is assumed to be a function of the distance of P from the center C, $\|P - C\|$, and thus fulfills the following properties.

Property 1. Given two generic points P_1 and P_2 of the circular area, the following implications hold:

$$\|P_1 - C\| < \|P_2 - C\| \Rightarrow \delta(P_1) > \delta(P_2) \tag{4.1}$$

$$\|P_1 - C\| = \|P_2 - C\| \Rightarrow \delta(P_1) = \delta(P_2) \tag{4.2}$$

Property 2. Given two generic points S_1 and S_2 where two sensors will be/are placed, the following holds:

$$\|S_1 - S_2\| \leq \gamma\rho \tag{4.3}$$

with $\gamma \in [0, 1] \subset \mathcal{R}$ being a *safety factor* (usually set to about 0.9), used to scale the maximum allowed distance to a lower (and thus safer) value than the transmission range ρ.

The following section describes an algorithm to plan a proper path to be followed by the UAVs: the algorithm is designed to ensure the properties described above and also to minimize the mission time.

4.2 Single Area Coverage for Sensor Deployments and path planning

As stated in Section 4.1, sensors are to be released into an area of terrain composed by a number of "hot zones" modeled with a circular shape and a fixed radius. In this section, we explain the model designed to drive the UAVs over a single hot zone in order to perform sensor releasing. The model is based on two main design principles:

Non uniform sensor density. Sensors are released in such a way that, the smaller the distance from the center, the larger the density of released sensors (see Equations 4.1 and 4.2 in Section 4.1).

Overhead minimization. Based on the common practices of single area coverage, the path followed by the UAVs should be designed in order to minimize the total *overhead*, i.e. the ratio between the distance covered without releasing sensors and the overall path length.

In the following, we will use the notations described in Table 4.1.

The area partitioning scheme is designed to divide the area of terrain into equal circular sectors ("pie slices"), each of angle θ, that we may call **sections**. As shown in Figure 4.4, this leads to

R	the radius of the hot zone, modeled as a circular area
s	discretization factor, a fraction of the radius R
n_{rings}	number of rings defined by the step "s"
n_{sec}	number of sections
ρ	maximum communication range of the single sensor
γ	Communication range scaling factor
r_d	UAVs minimal reciprocal distance, for safety reasons
v_a	Speed of the single UAV
S_{tot}	number of sensors released in the area

Table 4.1: Symbol Table

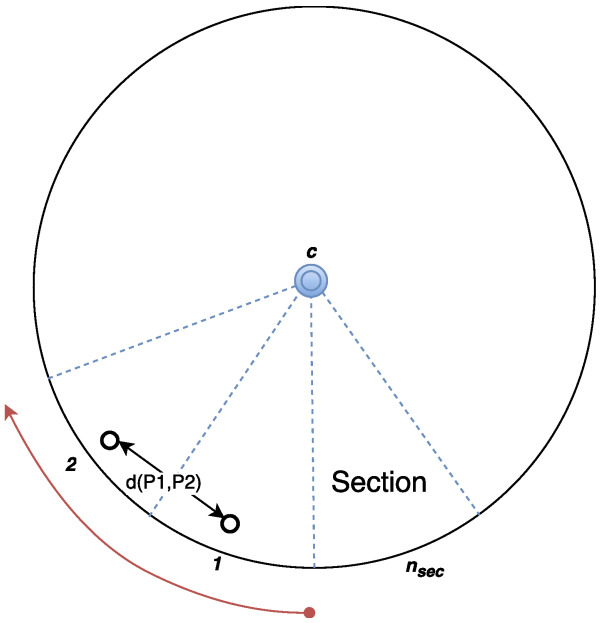

Figure 4.4: Single area partition into n_{sec} sections (the wedges between the dashed blue lines).

partitioning the area into n_{sec} sections, each one of width $\theta = \frac{2\pi}{n_{sec}}$. Each section is served by a single UAV, which releases sensors as it performs a movement from the outer border of the area towards the center, and back (see Figure 4.5).

All UAVs release sensors in the same manner, by following the same *path shape*, shown in Figure 4.5, as explained below:

1. each UAV begins its mission in a specific section, starting from the border of the area and releasing sensors as it moves towards the center of the area.

2. At the end of the previous motion, the UAV moves from the current section to one of the adjacent sections, in order to repeat the previous path, in the new section, in the opposite direction of travel (from the center to the border).

3. At the end of this motion, the UAV moves to the adjacent section and restarts a path according to point 1.

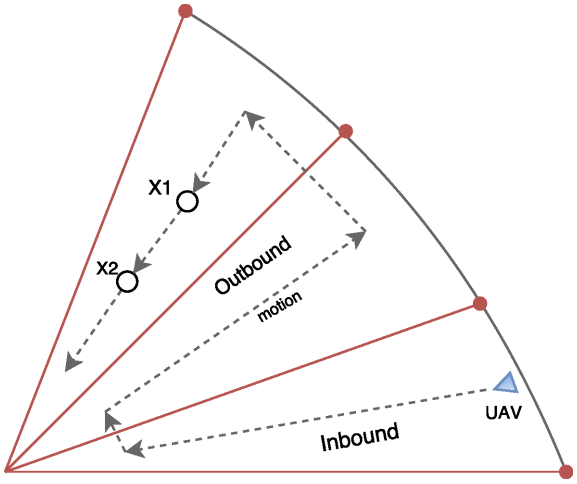

Figure 4.5: UAVs motion along sections

The overall path followed by each UAV is thus a concatenation of a series of *contiguous sub-paths* $p_1, p_2, \ldots, p_k$ where *odd* elements $(p_1, p_3, p_5, \ldots)$ are *inbound sub-paths*, i.e. from the border to the center of the area, and *even* elements $(p_2, p_4, p_6, \ldots)$ are *outbound sub-paths*, i.e. from the center to the border of the area. Since UAVs must eventually leave the area, and it would be wasteful not to have them lay sensors along their return path, it follows that each UAV path should contain as many outbound as inbound paths. As a consequence, the maximum number of UAVs that can be sensibly used to cover the area in a single mission is $\lceil \frac{n_{sec}}{2} \rceil$.

UAVs will follow their path in a synchronized manner, with identical speed and sensor release rate. Let P_1 and P_2 be the positions of two neighbor sensors released into two adjacent sections near the border of the area. Let $d(P_1, P_2) = \|P_1 - P_2\|$ be the distance between them (see Figure 4.4). Let us recall that devices are assumed to be of the same type, and that ρ indicates their maximum communication range, in meters, (see Section 4.1 and Table 4.1). Then, in order to ensure that every pair of neighbor sensors, lying close to the outer border of the area, will be able to communicate, the condition $d(P_1, P_2) \leq \overline{\rho} = \gamma\rho$ must hold, where $\gamma \in (0, 1)$. Parameter γ is introduced to obtain the tighter $\overline{\rho}$ bound, so as to increase the probability that, even in the face of adverse circumstances occurring after their release, sensors P_1 and P_2, will still be able to exchange data. In particular, it is easy to see (Figure 4.4) that the worst (maximum distance) case occurs when P_1 and P_2 lie *exactly* on the the border, so that (recalling $\theta = 2\pi/n_{sec}$): $d(P_1, P_2) = 2R\sin(\pi/n_{sec})$. Therefore, the following constraint must hold:

$$2R \sin\left(\frac{\pi}{n_{sec}}\right) \leq \gamma\rho \tag{4.4}$$

to ensure that any pair of neighbor sensors released into two adjacent sections will be able to communicate. Note that, for given R (depends on hot area size) and ρ (a characteristic of the sensors), this is actually a constraint requiring sections (pie slices) to be numerous enough.

Let us now denote with v_a the speed of the single UAV/agent, and with r_{dep} the rate of

deposition of the sensors. A strategy to obtain a variation in the sensor density, over the radius of the single area, could be to fix a member of the $\{v_a, r_{dep}\}$ parameter pair, while letting the other evolve, along the path of the UAV, in accordance with a suitable law.

In our model, we adopted a simple but flexible law for the variation over the path of the deposition rate r_{dep}, measured in *number of sensors (laid) per second*,:

$$r_{dep}(x) = \frac{a}{1 + be^{-cx}} \tag{4.5}$$

with a, b, c positive, and the variable x representing the distance from the border of the area. Thus, the deposition rate starts at $r_{dep}(0) = a/(1 + b)$ on the border, and steadily increases as the UAV approaches the center, so that x grows and e^{-cx} decreases. Parameter a represents the saturation value $r_{dep}(\infty)$, which will be approximated at the center, by $r_{dep}(R) = a/(1+be^{-cR})$, provided the radius R is "long enough" for the given values of b and c. Parameter c allows the designer to shape the steepness of the curve, i.e. how quickly the saturation value a is reached. Finally, parameter b is a scaling factor, enabling the starting rate, $r_{dep}(0) = a/(1 + b)$, to be tuned. Figure 4.6 shows a plot of equation 4.5 for $b = 1, a = 1$ and several values of c.

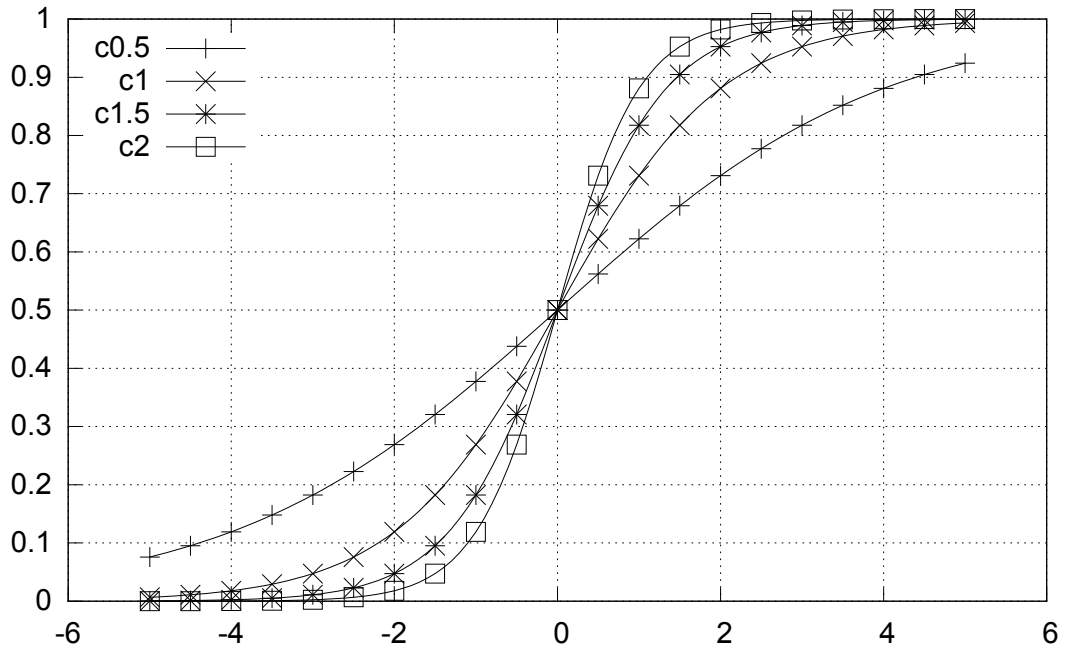

Figure 4.6: Function $\frac{a}{1+be^{-cx}}$ with $a = 1, b = 1$, and several values of c

Equation 4.5 represents a theoretical model (indeed the rate variation cannot be actually continuous). The *upper bound* on the total number of sensors released along the path from a distance $x = 0$ to a distance $x = \overline{x}$ from the *border of the area* by the single UAV can be computed as follows:

$$
\begin{aligned}
S(\overline{x}) = & \left\lceil \int_0^{\overline{x}} r_{dep}(x)dx \right\rceil = \left\lceil \int_0^{\overline{x}} \frac{a}{1+be^{-cx}}dx \right\rceil = \left\lceil \frac{a \cdot ln(e^{c\overline{x}}+b)}{c} - \frac{a \cdot ln(1+b)}{c} \right\rceil \\
= & \left\lceil \frac{a}{c}\big(ln(e^{c\overline{x}}+b) - ln(1+b)\big) \right\rceil = \left\lceil \frac{a}{c} ln \frac{e^{c\overline{x}}+b}{1+b} \right\rceil
\end{aligned} \tag{4.6}
$$

Recalling that, for security reasons, UAVs cannot be closer than r_d (see Table 4.1 and equation 4.8), and that they proceed synchronously towards the center, it follows that they cannot actually reach it, but must stop at a distance $R_{\max} < R$ from the outer border. It is easily derived that $R - R_{\max} = r_d/2sin(\theta/2)$, i.e. $R_{\max} = R - r_d/2sin(\theta/2)$.

Now, we can use equation (4.6) to compute $S(R_{\max})$ as an upper bound for the number of sensors released over a section, and hence obtain $n_{sec}S(R_{\max})$ as an excess estimation for S_{tot}, the total number of sensors available for the whole area.

During the flight of the UAVs, while the deposition rate varies as dictated by Equation 4.5, speed v_a is maintained steady for the whole trip of the UAV, which simplifies the management of the UAVs' control system. A suitable speed should be selected to avoid stressing the engines.

A suitable upper bound to set the positive parameter b in Equation 4.5 can be established as follows.

First of all, in order to guarantee that two sensors released in succession (see Figure 4.5), at positions x_1 and x_2, will be able to communicate, they should not be too far apart, i.e. the following condition must hold: $d(x_1, x_2) \leq \gamma\rho$. Now, this constraint is most difficult to satisfy at the outer edge, where sensor density is lowest. Assume that initially, at time 0, a sensor is dropped on x_1, on the border circle. For the UAV to move to x_2 along the radius, at speed v_a, and drop the 2nd sensor, the time required is $d(x_1, x_2)/v_a$. Thus, we get, for the initial drop rate $r_{dep}(0) = 2/(d(x_1, x_2)/v_a) = 2v_a/d(x_1, x_2)$, whence $d(x_1, x_2) = 2v_a/r_{dep}(0)$, and, recalling the constraint $d(x_1, x_2) \leq \gamma\rho$:

$$2v_a/r_{dep}(0) \leq \gamma\rho, \text{ so: } \quad r_{dep}(0) \geq 2v_a/\gamma\rho$$

Moreover, in the model adopted, $r_{dep}(0) = a/(1+b)$, yielding the inequality:

$$r_{dep}(0) = \frac{a}{1+b} \geq 2\frac{v_a}{\gamma\rho}, \text{ so: } \quad b \leq \frac{a}{2v_a}(\gamma\rho) - 1 \tag{4.7}$$

Finally, parameter c will allow the operator to plan the mission by tuning the steepness of the deposition rate curve.

Equation 4.5 represents a theoretical model and must be applied to instruct the UAV to release sensors during their paths. To this aim, a **ring** is defined as the circular area of width s as shown in the left side of Figure 4.7. The number of rings is defined by fixing the step s, whence $n_{rings} = \lfloor \frac{R}{s} \rfloor$. The areas resulting from the intersection of the rings with the sections are called **sectors** (see right side of Figure 4.7). Introducing rings is useful to place into a real scenario the deposition rate law defined by Equation 4.5.

Once the speed of the UAV is set to v_a, the sensor deposition rate will vary stepwise, every

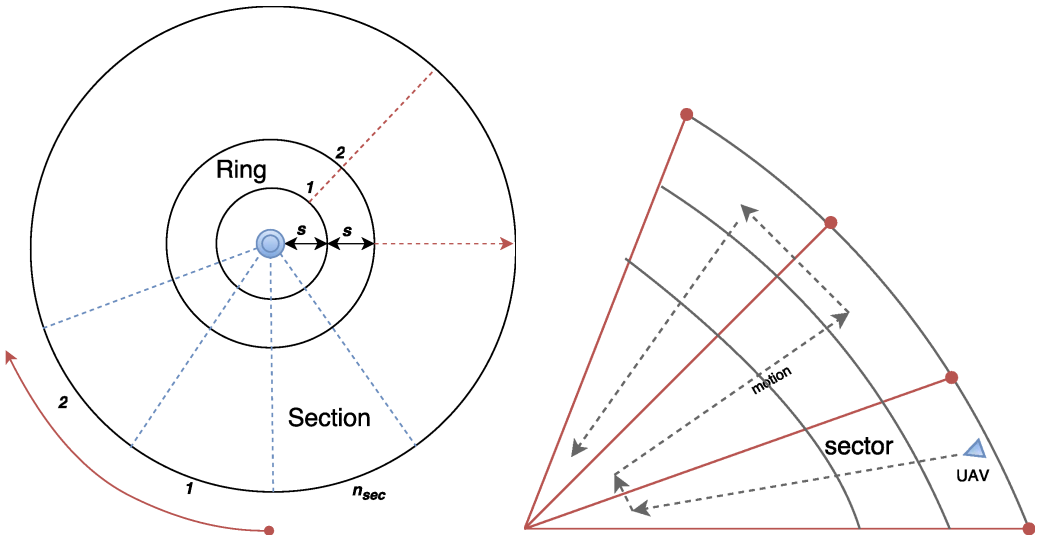

Figure 4.7: Left: rings, sections, sectors. Right: UAV paths trough sectors

time the UAV's current position is incremented by s meters. In particular, along the path from point x_1 to point $x_2 = x_1 + s$, the deposition rate of the single UAV is fixed to $r_{dep}(x_1)$.

Furthermore, it must be recalled that UAVs have to respect a reciprocal security distance r_d, representing the minimum required level of operational safety to avoid collisions. Therefore, for two UAVs that are moving over two adjacent sections and have reached the innermost ring, it must be the case that:

$$r_d \leq 2s \sin\left(\frac{\pi}{n_{sec}}\right) \tag{4.8}$$

While condition (4.8) guarantees operational safety, (4.4) ensures connectivity among neighbor sensors. Indeed, they provide the means to determine suitable values of s (step length, i.e. number of rings) and n_{sec} (number of sections), respectively. Moreover, since $s < R$, it follows that $2s \sin\left(\frac{\pi}{n_{sec}}\right) \leq 2R \sin\left(\frac{\pi}{n_{sec}}\right)$, whence, by (4.8) and (4.4), $r_d \leq \gamma\rho$.

Based on the approach discussed above, we report here a numerical example set by taking into account an area with a radius of 2 kilometers. All the parameters are summarized in Table 4.2.

Sensors	Range of communication ρ	50 m
	Scaling factor γ	0.9
	Scaled range of communication $\overline{\rho}$	45 m
Sensor releasing (r_{dep})	v_a (speed)	{20, 25, 30}km/h
	Maximum sensors deposition rate (a)	1/s
	b	1.5
	Steepness for deposition rate (c)	$[5 \times 10^{-6}, 5 \times 10^{-5}]$
Area	Radius (R)	2000 m
	No. of sections (n_{sec})	279
	Step s	$\overline{\rho}$
	No. of rings (n_{rings})	44 ($\frac{r}{\overline{\rho}}$)

Table 4.2: Sample parameters for planning a mission in a single area

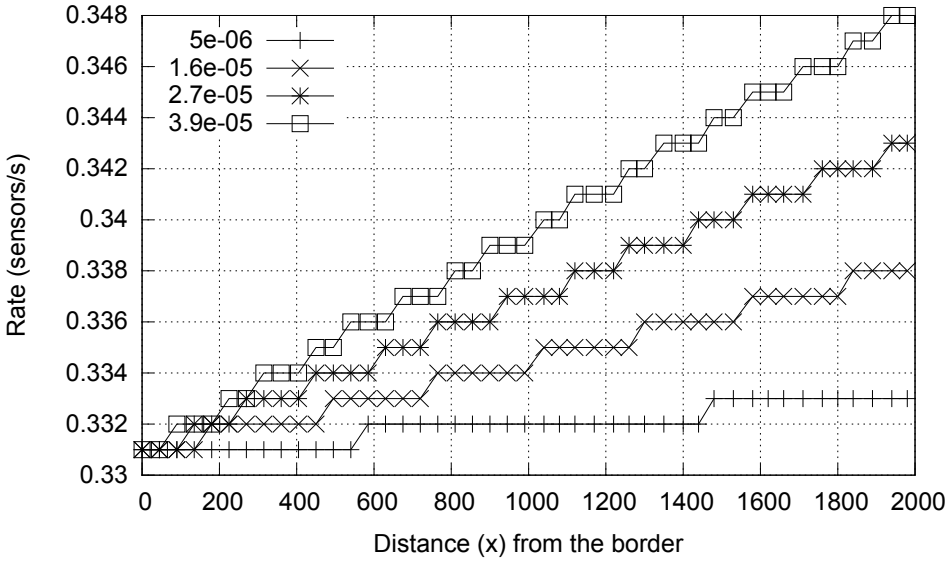

Figure 4.8: Deposition rate

	$v_a = 20km/h$		$v_a = 25km/h$		$v_a = 30km/h$	
c	S_{tot}	$\frac{S_{tot}}{n_{sec}}$	S_{tot}	$\frac{S_{tot}}{n_{sec}}$	S_{tot}	$\frac{S_{tot}}{n_{sec}}$
$5 \cdot 10^{-6}$	150k	540	190k	660	220k	790
$16 \cdot 10^{-6}$	150k	540	190k	670	220k	790
$27 \cdot 10^{-6}$	150k	550	190k	670	220k	800
$39 \cdot 10^{-6}$	150k	550	190k	680	220k	800
$5 \cdot 10^{-5}$	160k	560	190k	680	230k	810

Table 4.3: Total number of sensors and number of sensors per sections, different speed and steepness (c)

The sensor communication range ρ, along with a scaling factor γ, has also been employed to set the width of rings. UAV speed ranges over a set of three values, and the maximum deposition rate (essentially parameter a in Equation 4.5) is set at 1 sensor per second; parameter b is computed in accordance with Equation 4.7. Finally, the steepness parameter c ranges from 5×10^{-6} to 5×10^{-5}.

Figure 4.8 shows the evolution of the sensor deposition rate for a speed of 25km/h and 4 different values of c.

Finally, Table 4.3 summarizes the total number of sensors that will be released in the whole area — computed as in formula 4.6 —, and the number of sensors per section $\frac{S_{tot}}{n_{sec}}$.

4.3 Cluster and Task Organization

The scenario introduced in the previous section is illustrates in Figure 4.9 in order to show the organization of the JarvSis nodes hosted on the base station and UAVs. In particular a JarvSis

node is deployed in the base station and a different JarvSis node is deployed on each UAV. The nodes are using an identical software version but are characterized by a different Cluster and Task structure. The particular JarvSis network formed by these nodes is not hierarchical and all the tasks and clusters are defined within nodes at the same hierarchical levels. As a consequence, the behavior of the nodes will be not subject to any other node/entity of a higher hierarchical level.

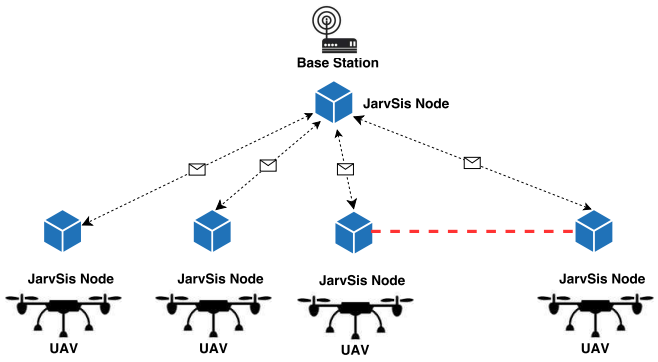

Figure 4.9: UAV and Base Station JarvSis nodes

A JarvSis node running into a UAV hosts 4 different clusters of tasks; each cluster, in turn, represents a single JarvSis task, as already explained in Section 2 and illustrated in Figure 4.10. In particular, the *starting cluster* represents the task *Release Rate*, which is responsible to send sensor a release messages accordingly to the release rate set by the user (Figure 4.11).

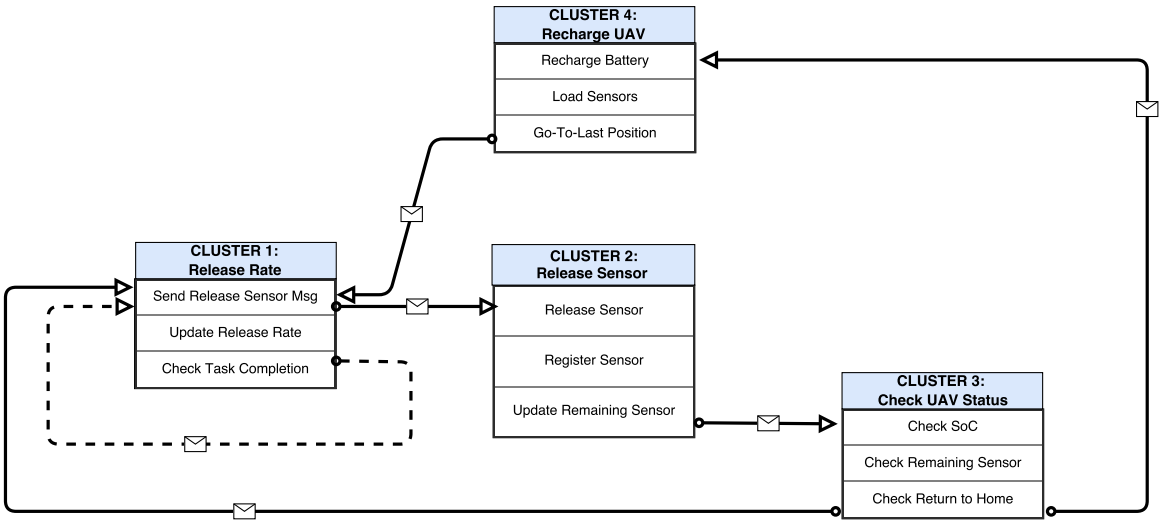

Figure 4.10: UAV JarvSis node Cluster/Tasks

The required actions are executed by the the three tasks that compose the cluster – these are executed sequentially – by these tasks the following actions are executed:

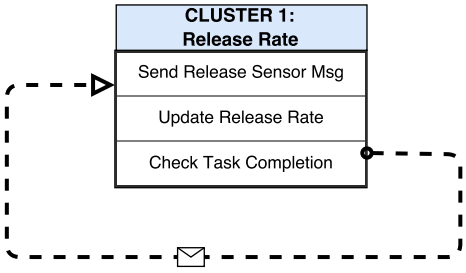

Figure 4.11: Release Rate Cluster/Tasks

1. Send a start message to the "Release Sensor" cluster to allow a sensor release.
2. Update the sensor release rate, set the timeout between one sensor release and the next one and wait for the selected timeout before start the next task.
3. Check if the final position is reached before the next sensor release.

As described in Figure 4.11 the execution of the tasks of the cluster is repeated until the final position is reached or the UAV comes back to home to recharge the battery or to fill the sensors container. The sensor release is controlled by the "Release Sensor" cluster that is composed by the following three tasks (Figure 4.12):

1. Release the sensor.
2. Register the sensor on the base station JarvSis node with a specific start message to the "Process UAV Msg" cluster.
3. Update the number of the remaining sensors according to the number of released sensors.

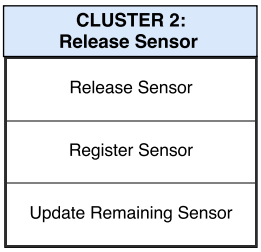

Figure 4.12: Release Sensor Cluster/Tasks

As illustrated in Figure 4.13, the cluster `Release Sensors` represents the connection between the JarvSis node running into the base station and the UAV JarvSis node. The message is sent by the `Register Sensor` task through the MQTT broker hosted within the base station. This means that the UAV must be capable to connect to the base station that must act as a wireless hot-spot for each UAV.

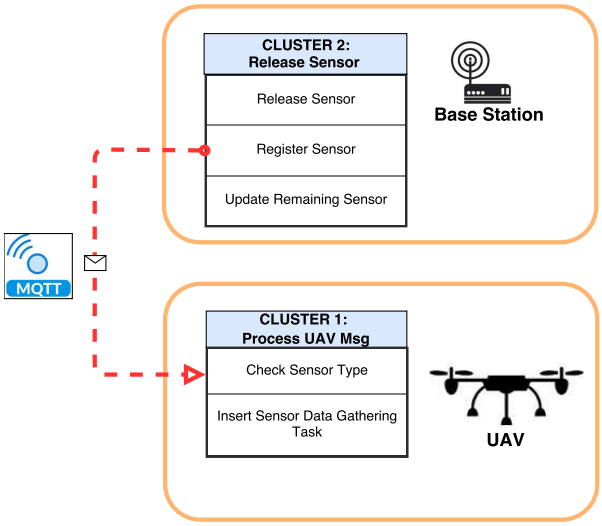

Figure 4.13: UAV and Base Station connection point

After each sensor release, the UAV must run a partial check of the system to ensure that is capable to continue the mission with the current amount of residual energy and sensors available in the container. To perform these actions a sequential cluster `Check UAV Status` is defined (Figure 4.14). Also in this case the tasks of the cluster are executed sequentially. The tasks are the following:

1. Check the PS (Power State) of the UAV battery according to the current mission.
2. Check the residual sensor amount on the UAV sensor container according to the current mission.
3. Based on the results of the two previous tasks, check if the UAV must return to its starting position to restore the PS or to fill the sensors container.

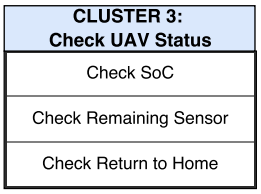

Figure 4.14: Check UAV Cluster/Tasks

In case the UAV has to come back to its starting position, it will send a *stop message* to the `Release Rate` cluster and will lead the UAV to the home position. Once the starting Position is reached, the UAV will restore the PS and the sensors in the container and will continue the mission.

To this end the `Recharge UAV` cluster is defined (Figure 4.15), as follows:

1. Restore the PS of the UAV battery.
2. Load the sensors into the container.
3. Return to the previous position or wait for another mission if the previous one was completed.

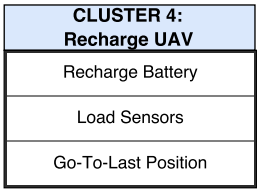

Figure 4.15: Recharge UAV Cluster/Tasks

Even if the tasks cannot be automated and must be performed manually by a human operator, they must be needed to check and confirm that the UAV is ready to complete the existing mission or start a new one.

The Base Station hosts a JarvSis node which is composed by 9 connected clusters of tasks (Figure 4.16), that can be classified into three different logical *sections*, as follows:

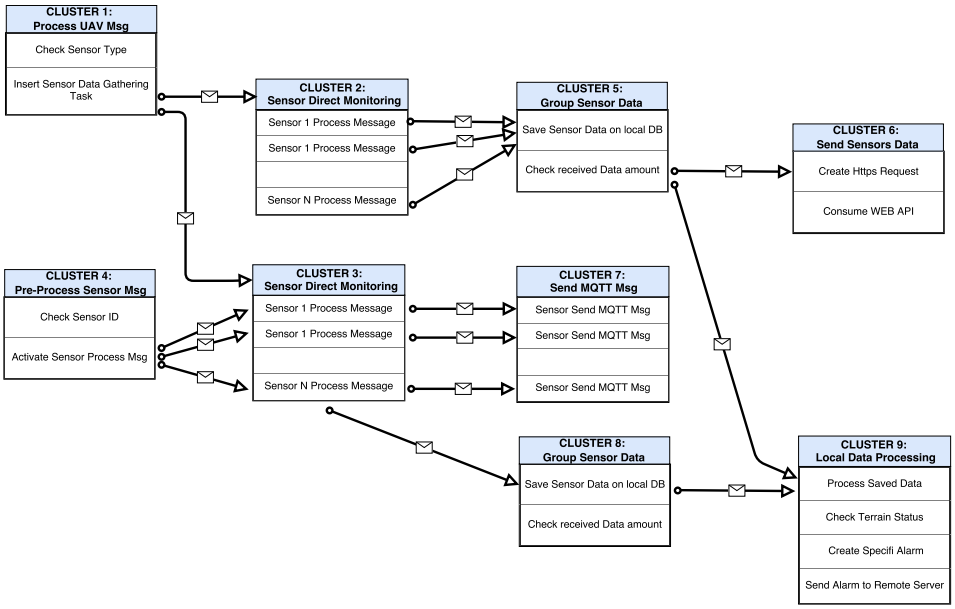

Figure 4.16: Base Station JarvSis node Cluster/Tasks

- *Sensors registration* within the JarvSis node and sensors data pre-processing. The clusters that compose this section are used to register the sensors within the JarvSis node and pre-process all the data from the field before the transmission to a remote application. Through

this section the JarvSis node clusters/tasks structure may evolve dynamically according to the sensors already deployed. Furthermore all the operations on data may enable to map sensors data to different standards required by a local or a remote application. This section is composed by the following clusters (Figure 4.17):

1. **Process Uav Msg** – it is the "starting cluster" where messages coming from the UAVs are processed – this cluster is responsible to analyze the data received from the UAV, identify the sensor type and register the sensor in the right cluster (Sensor Direct Monitoring). Through the Adaptive Scheduler interface the cluster is able to modify at run-time the structure of another cluster of the same node. This functionality is then the key to adapt dynamically the JarvSis node to the environment.
2. **Sensors Direct Monitoring**: two different clusters are defined to monitor and process the data coming from the sensors on the field (accordingly to the two main sensors types). These clusters (Cluster 2 and 3 in Fig 4.17) will convert the incoming data to a proper format before the transmission to a remote application. In case of sensors capable to send MQTT message to the base station the cluster can forward the message payload directly to a remote broker/WEB-API or convert the message payload to another format and then forward the data.
3. **Pre-Process Sensor Msg**: in case of sensors organized as a mesh network and send data to the base station, a pre-processing stage is required.

   While in case of MQTT message received JarvSis is capable to map directly a received message to a specific task through the sensor id, the direct mapping it's not possible if the message received is in a different format. Then through a specific interface the data must be processed, and forwarded to the specific task/cluster through the sensor id. This cluster is then used to process the data that comes from the local mesh network and forward the data to the specific task/cluster accordingly to the sensor id.

- *Sensors data transmission to a remote software platform.* Since the base station must be able to send the sensors data by means of i) an external MQTT broker and a ii) dedicated RestFul Web API, the MQTT message that comes from the sensors and that are grouped and sent to a RestFul WEB-API while the data received from the local mesh network are converted to JSON and sent as an MQTT message payload to an external broker. The section is composed by the following clusters (Figure 4.18):

  1. **Group Sensor Data**: The MQTT message received by the sensors are saved on a local Sqlite database [52] until a certain size is reached, then the data will be sent to the remote application through the exposed RestFul WEB-API.
  2. **Send Sensor Data**: After the grouping phase the data saved on the local database can be sent to the remote application with the consuming of the dedicated RestFul

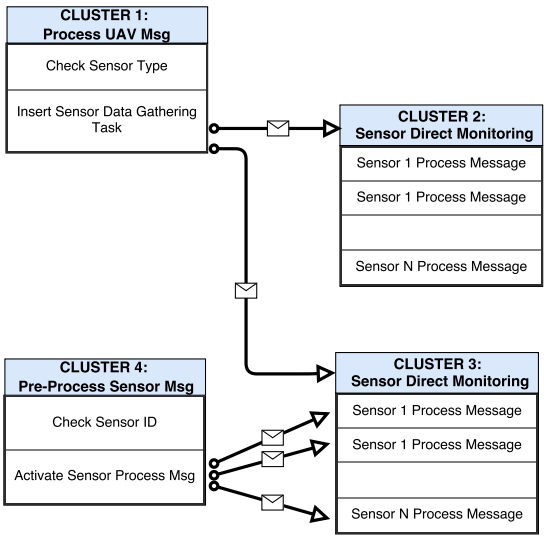

Figure 4.17: Sensor Registration and data pre-processing Cluster/Tasks

WEB-API. A Json is created at run-time (that contain the data array) according to the required format and an HTTP/HTTPS request is generated from JarvSis following the WEB-API speficication. JarvSis is able to perform this action through the HTTP request manager that is able to perform multiple HTTP/HTTPS request at runtime according to different tasks needs and format.

3. **Send MQTT Msg**: After the pre-processing phase the data can be encapsulated in a MQTT message and sent to a remote Broker in a specific topic. JarvSis is able to perform this action through the MQTT client manager that can manage multiple Broker and Topics at same time.

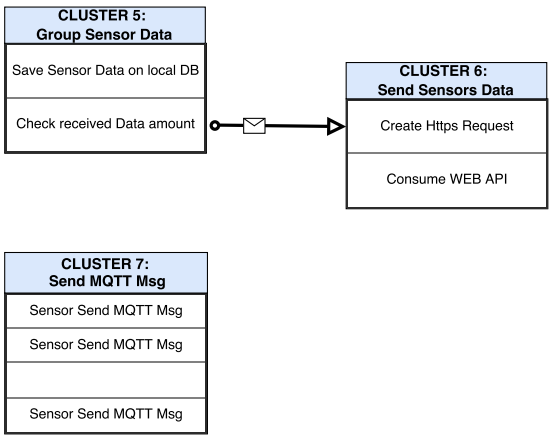

Figure 4.18: Sensors data remote transmission Cluster/Tasks

- *Sensors data processing by custom application.* In an increasing number of applications and architecture there is the need to perform some data processing locally without using the cloud/remote resource. As discussed in Section 2, JarvSis is actually designed to construct FOG platforms [69] with a scalable architecture, and it is capable to store complex processing capabilities according to the available computational resources. To describe this capability a simple local application is implemented within JarvSis through the usage of two dedicated clusters. The section is composed by the following clusters (Figure 4.19).

  1. **Group Sensor Data**: This cluster is responsible to collect a specific amount of data from the sensors on filed and save the data on the local Sqlite database. The data is then used by the "Local Data Processing" cluster once that the received data amount is enough.
  2. **Local Data Processing**: The data gathered from sensors on field are processed to evaluate the terrain status. If the terrain status can be classified as "critical" (accordingly to a specific algorithm or thresholds) then an specific alarm is created and sent to a remote monitoring platform through a RestFul WEB-API. To improve the overall redundancy the alarm can be sent to multiple remote end-points with a specific fallback algorithm that can be implemented within an internal JarvSis task or an external platform available in the local area network.

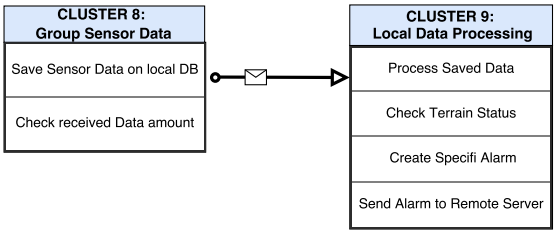

Figure 4.19: Local Application Cluster/Tasks

The whole structure composed by the clusters defined among the JarvSis nodes previously described is represented in Figure 4.20. The Figure describe only the connection between a single JarvSis UAV node and the JarvSis base station node, but this schema can be extended to all the UAV JarvSis nodes. Even if there is a single connection point between the UAVs and the base station the usage of the MQTT protocol prevent the introduction of possible bottleneck also in if a relative high number of UAV is used to deploy the sensors on the target areas.

4.4 Simulation of IoT Sensors and Platform Integration

To simulate the sensors deployment and the data transmissions from sensors to remote application a dedicate environment was created. In order to simplify the simulation, only a single UAV,

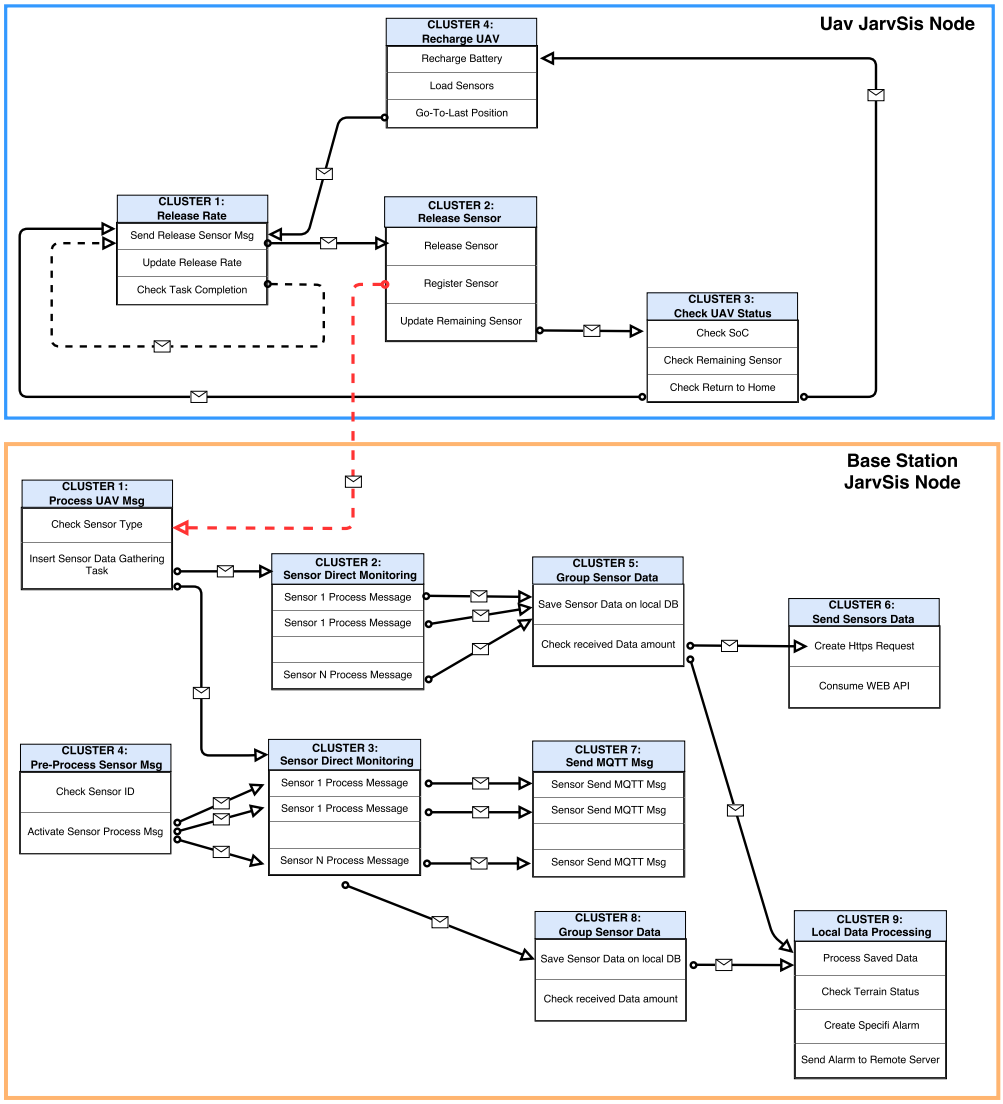

Figure 4.20: UAV and Base Station JarvSis node Cluster/Tasks

with the related JarvSis node, has been simulated. The complete simulation includes also data transmission, and the scenario is composed by the following elements:

- Two JarvSis nodes used to simulate a single UAV and the base station.
- Two MQTT Broker: the first one is used as a *local* broker where the two JarvSis nodes can exchange message and also the sensors can publish their data, the second one is used as a remote broker where data sensors (that come from the local mesh network) must be published.
- A node-js RestFul WEB-API that simulate a remote application that can receive the sensors grouped data.

- A web-socket java server used to simulate the data sent by the sensors that are not able to send MQTT messages.
- An AgentSimJs based web portal (see Chapter 3) used to configure the sensors deployment and to simulate the releasing phase and the data generation and transmission by each single sensor.

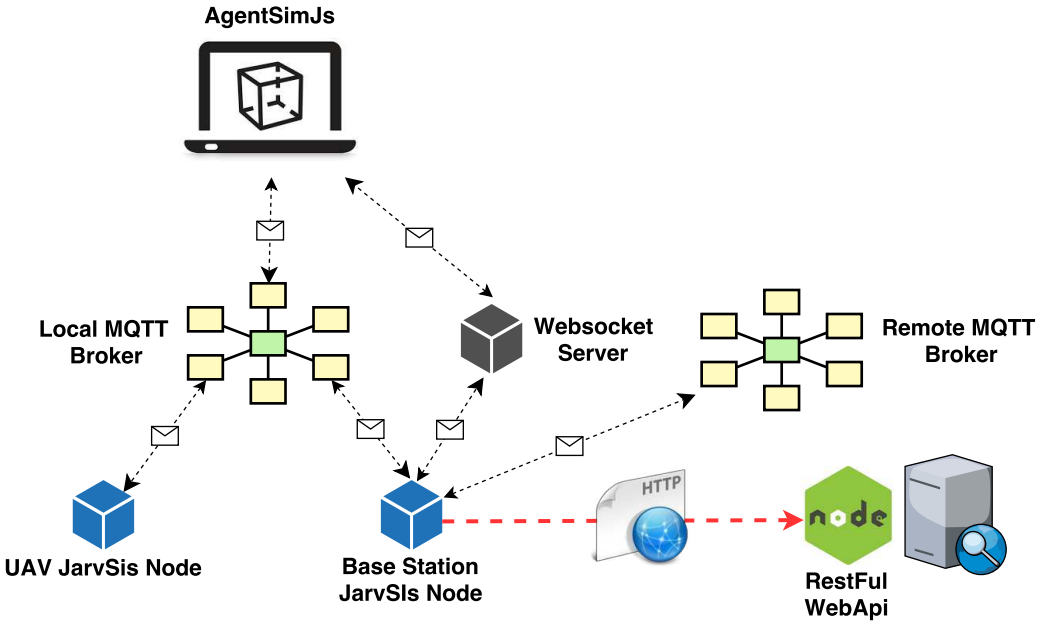

Figure 4.21: Simulation Environment Overview

The UAV is simulated as an agent within AgentSimJs environment. It will receive the MQTT message by the JarvSis node to execute the external tasks related to sensors deployment and path following. The JavaScript function that create sectors and section includes also the generation of UAVs spawn point and the path planning for the sensors deposition along the assigned sections of the target area.

The parameters related to the area definition can be customized by the user through a dedicated web interface before running the simulation (Listing 4.1).

Listing 4.1: Circular Area Decomposition and UAV path Planning

```
var uav_number = $("#uav_number").val();
          //circular area decomposition
      var sector_x_uav=Math.floor(number_of_sectors/uav_number);
          angle = 2*Math.PI/ number_of_sectors;
          uav_array_sims=[];

         for (var i=1; i<= uav_number;i++){
                var uav = new Uav(i,1,2);
                uav.setSectorNumber(sector_x_uav);
                uav_array_sims.push(uav);

         }

      var number_of_s_approx= sector_x_uav*uav_number;

      var dt_sector=number_of_sectors-number_of_s_approx;

```

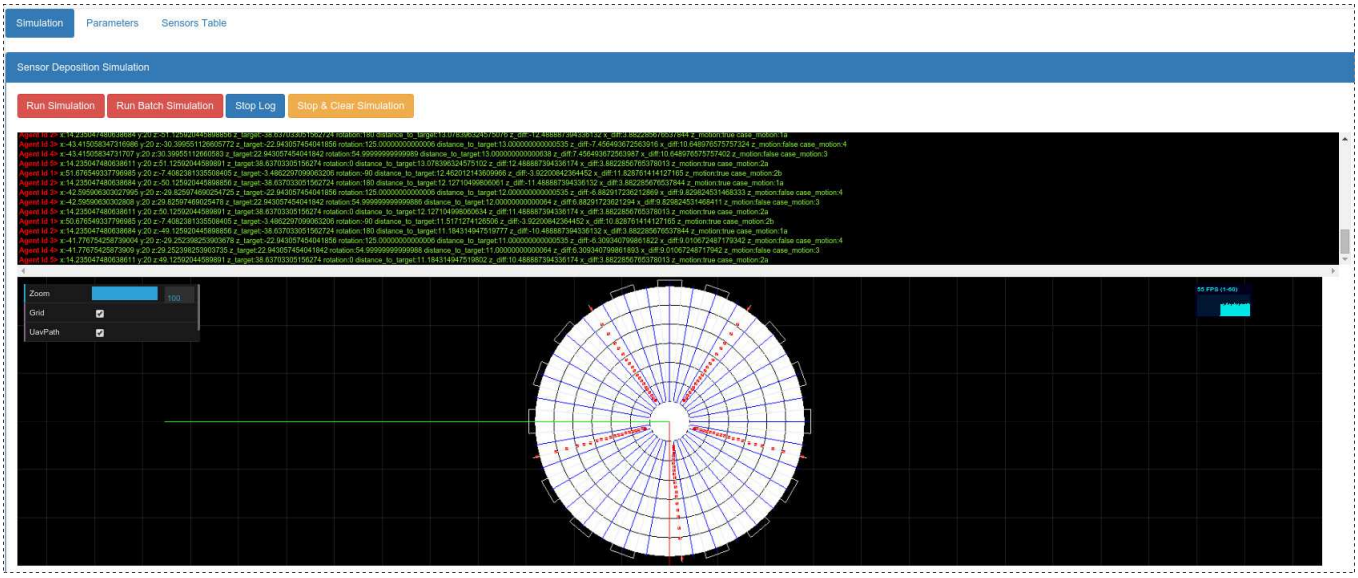

Figure 4.22: Sensor Deployment Simulation

```
18        //unassigned vector re-distribution
19        if (number_of_sectors>number_of_s_approx){
20
21                  for (var i=0; i< uav_array_sims.length;i++){
22                       if(dt_sector>0){
23                            var previous_sector_number=uav_array_sims[i].number_of_sector;
24                            var new_sector_number=previous_sector_number+1;
25                            uav_array_sims[i].setSectorNumber(new_sector_number);
26                            dt_sector=dt_sector-1;
27                       }
28                  }
29
30
31        }
32
33        //uav spawn point - path planning and representation
34        var sect_counter=0;
35        for (var i=1; i<= uav_array_sims.length;i++){
36
37                   var sector_x_uav=uav_array_sims[i-1].number_of_sector;
38                   sect_counter=sect_counter+sector_x_uav;
39                   var starting_angle=angle*(sect_counter+1);
40
41                   var spawn_point=[{x:Math.sin((angle*(sect_counter+1)) +angle/2 )*radius_area,
42                       y:20, z:Math.cos((angle*(sect_counter+1)) +angle/2 )*radius_area}];
43                    uav_array_sims[i-1].spawnAt(Math.sin((angle*(sect_counter+1)) +angle/2 )*
44                            radius_area, 20, Math.cos((angle*(sect_counter+1)) +angle/2 )*radius_area,
                                 scene_sims);
45                    uav_array_sims[i-1].setDivLog("simulation_log");
46
47
48                    var sector=sector_x_uav;
49                    var sect_angle=angle;
50
51                    var path_uav=buildPath(sector_x_uav,starting_angle,sect_angle);
52                    uav_array_sims[i-1].setPath(path_uav);
53                    uav_array_sims[i-1].setSpawnPoint(spawn_point);
54
55                    plotPath(path_uav,spawn_point,uav_array_sims[i-1].id);
56
57
58    }
```

To create the path of each UAV a specific function is used (Listing 4.2) that take as input the UAV related sector, the starting angle of the section and the section angle will return a complete path composed by multiple back and forward segments connected by small transition segments. During the back and forward segments the UAV will proceed with the sensor release while during the transition phase the sensors deployment will be inhibited.

Listing 4.2: Build Path Function

```
function buildPath(sector_x_uav,starting_angle,section_angle){

        var path=[];
        var center_pos=true;



        for (var j=0; j<sector_x_uav;j++){

            if(center_pos==true){

                path.push({x:Math.sin(starting_angle + section_angle*j + (section_angle/2) )*(
                    radius_area/number_of_levels)
                    , y:20,
                    z:Math.cos(starting_angle + section_angle*j + (section_angle/2) )*(radius_area/
                        number_of_levels),center_point:true});


                if ((j+1)<sector_x_uav) {

                    path.push({x:Math.sin(starting_angle + section_angle*(j+1) + (section_angle/2) )
                        *(radius_area/number_of_levels)
                    , y:20,
                    z:Math.cos(starting_angle + section_angle*(j+1) + (section_angle/2) )*(
                        radius_area/number_of_levels),center_point:true});

                }

                center_pos=false;
            }else{
                path.push({x:Math.sin(starting_angle + section_angle*j + (section_angle/2) )*(
                    radius_area)
                    , y:20,
                    z:Math.cos(starting_angle + section_angle*j + (section_angle/2) )*(radius_area),
                        center_point:false});

                if ((j+1)<sector_x_uav) {
                    path.push({x:Math.sin(starting_angle + section_angle*(j+1) + (section_angle/2) )
                        *(radius_area)
                    , y:20,
                    z:Math.cos(starting_angle + section_angle*(j+1) + (section_angle/2) )*(
                        radius_area),center_point:false});

                }

                center_pos=true;


            }

        }



        return path;


}
```

To facilitate the simulation of different scenarios with several area decomposition parameters a batch simulation worker was implemented (Figure 4.23). Through this worker the user is able to change the area decomposition parameter and deploy the sensors on the AgentSimJs environment without waiting the path completion. This approach allow also to deploy an higher amount of sensors and to simulate multiple UAVs that deploy sensors simultaneously.

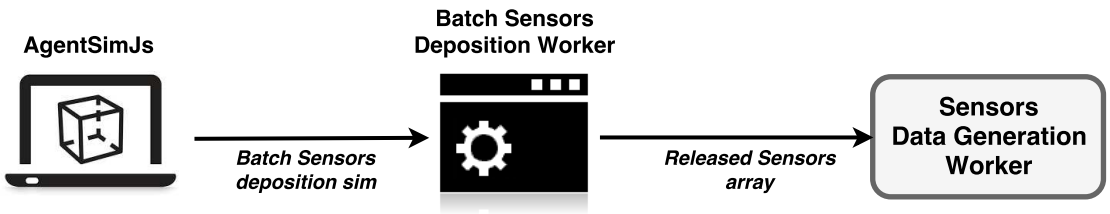

Figure 4.23: Batch Simulation Flow

Listing 4.3: Sensor Deployment Batch Simulation

```
self.addEventListener('message', function(e) {
        sim_par=e.data;


        RunBatchSim(sim_par[0].max_speed,sim_par[0].c_par,sim_par[0].a_par,sim_par[0].sensor_range,
             sim_par[0].scaling_factor,sim_par[0].nl,sim_par[0].ring_step,sim_par[0].current_sector,
             sim_par[0].sensor_id);

}, false);

function RunBatchSim( max_speed_in,c_par_in,a_par_in,sensor_range_in,scaling_factor_in,nl_in,
     ring_step_in,current_sector,sensor_id){
        console.clear();
        var max_speed=max_speed_in;
        var c_par =c_par_in;
        var a_par =a_par_in;
        var scaling_factor=scaling_factor_in;
        var sensor_range=sensor_range_in;
        var b_par =(a_par /max_speed)*scaling_factor*sensor_range - 1;
        var nl=nl_in;
        var ring_step=ring_step_in;
        var radius= nl*ring_step;
        var t_range = radius/max_speed;
        var t_delta=ring_step/max_speed;
        t_range=t_delta*(nl-1);
    var sensor_id=sensor_id;
        var sensors_released=[];
        var  number_of_sectors = Math.ceil(2* Math.PI* radius / sensor_range);
        var deposition_rateEvolution_real=[];

            console.log("a:" + a_par);
            console.log("b:" + b_par);
            console.log("c:" + c_par);
                console.log("scaling_factor:" + scaling_factor);
                console.log("max_speed:" + max_speed);
                console.log("sensor_range:" + sensor_range);

                self.postMessage("Starting Batch Simulation..</br>Evaluating sensor rate deposition
                     Evolution..");
            //evaluate sensors deposition rate evolution accordingly to the model
            for (var i=0;i<t_range;i +=t_delta){

                 var rate_v= a_par/(1+b_par*Math.exp(-c_par*i));
                        self.postMessage("Sector: " + (i+1)/t_delta + " rate value: " + rate_v);
                        deposition_rateEvolution_real.push(
                                        {
                                                s: i,
                                                rate_value: rate_v,
                                                dt: t_delta
                                        });
                        console.log("rate_v:" + rate_v);


                }

                self.postMessage("Starting sensors releasing simulation..");
                var angle = 2*Math.PI/ number_of_sectors;
                self.postMessage("Number of Sectors:" + number_of_sectors);

                //simulate the sensor deposition along the areas
                for(var j=0;j<deposition_rateEvolution_real.length;j++){

                                        var dt_dep= 1/deposition_rateEvolution_real[j].rate_value;
                                         console.log("dt_dep:" + dt_dep);
                                        self.postMessage("dt_dep:" + dt_dep);

                                        var starting_point=0;
                                        if(j>0){
                                                starting_point=(deposition_rateEvolution_real.length -j)
                                                     * ring_step;
                                        }else{
                                                starting_point=(deposition_rateEvolution_real.length -j)
                                                     * ring_step;
                                        }

                                        self.postMessage("Section Starting Point :" + starting_point);

                                        for (var i=dt_dep;i<=t_delta;i+=dt_dep){

                                                        var x=Math.sin((angle*current_sector) +angle/2 )
                                                             * (i*max_speed + starting_point);
                                                        var y=20;
                                                        var z= Math.cos((angle*current_sector) +angle/2
                                                             )* (i*max_speed + starting_point);

                                                        sensors_released.push({
                                                                x: x ,
                                                                y: 20 ,
                                                                z: z ,
                                                                sensor_id:sensor_id,
                                                                sensor_ready: true,
                                                                current_sector: current_sector,
                                                                total_sector:number_of_sectors
                                                        });

                                                        var msg="Current Sector :" + i/dt_dep + "</br>
                                                             Current Section :" + current_sector + "</br
                                                             >" +
                                                        "Traveled distance: " + (i*max_speed) + "</br>"
                                                             +
                                                        "Releasing sensor " + sensor_id  + " at x:" + x
                                                             + " y:" + z + " z:" + y ;
```

```
91                                                  self.postMessage( msg);
92                                                  sensor_id=sensor_id+1;
93
94
95                                          }
96
97                              }
98
99                              var local = new Date();
100                             var localdatetime = local.getHours() + ":" + local.getMinutes() + ":" + local.
                                    getSeconds();
101                             self.postMessage("Section " + current_sector + " Simulation Completed at " +
                                    localdatetime+ "!");
102                             self.postMessage(sensors_released);
103                             close();
104
105                             //populate the sensors worker with the released sensors
106                             populateSensorWorker(sensors_released)
107
108 }
```

To simulate each single sensor a web worker (`Sensor Data Generation Worker`) is used (Figure 4.24). This worker takes as input an array with all the information related to the generated sensors (position,id etc.) and simulates the data generation of each single sensor. The worker characterize randomly some sensors as MQTT sensors (capable to sent MQTT message directly) or Web Socket sensors (that use Web Socket to sent data to the Base Station JarvSis node). The data is randomly generated but normalized within a certain range to simulate a different sensors type. Through a FOR loop that runs with a customizable timeout the randomly generated data is sent to the Base Station JarvSis node. The data is sent by using two different channels: MQTT broker and WebSocket. The MQTT Manager of AgentSimJs is used as an MQTT Client that receive data from the Sensor Data Generation Worker and push this data to the MQTT broker message bus. A WebSocket client was created from scratch and used to receive data from Sensor Data Generation Worker and push this data to the Web Socket Server where also the Base Station JarvSis node is connected. The whole process previously described is represented in Figure 4.18.

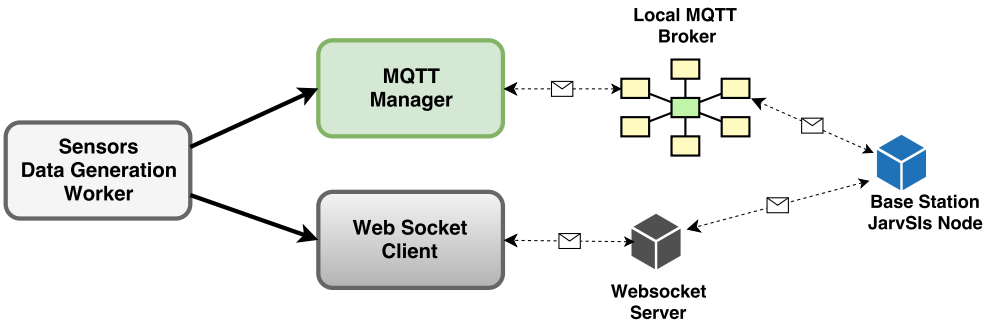

Figure 4.24: Sensor Data Generation Worker flow

In this use-case JarvSis was used as a local aggregation platform capable to gather the data from different sensors and send this data to a remote application. This kind of scenario is widely common in all the the application that require a huge amount of sensors or where different type of sensors were deployed in several phases. The presence of heterogeneous sensors often can implies the upgrade/substation of them to proceed with a more suitable integration. Through JarvSis, with a small effort on the development of a tiny interface between the sensors and JarvSis itself,

all the available sensors can be integrated and used by the same platform. Furthermore in this use case JarvSis was used also to perform some local data-processing directly on a JarvSis node. This capability can be used in mission-critical task or in specific scenario that requires low latency in data transmission and an high level of reliability. The tasks customization capability of JarvSis can also introduce the possibility to manage a local "eco-system" of sensors and devices that can be different in every scenario. Finally JarvSis was used to coordinate the sensors deployment and their connection with the related JarvSis node for the data gathering and remote transimission. The capability to integrate different platform/devices and hardware is fundamental to enable the automation of several processes that until now are fully manual and require a huge effect and time. JarvSis approach can be a enabling-key to the automation of this kind of scenario where autonomous robots will perform several tasks in an optimized and integral way through the usage of robots-cooperation techniques. This kind of application and the capability of JarvSis in this environment will be described in the next chapter where a set of heterogeneous robot will be used to perform a pre-defined set of tasks in a cooperative way.

5

Case study: JarvSis as a Multi platform manager for a multi-robot application

A difficult topic regarding cooperation and interoperability along different robotic platforms is represented by the communication and task management among different robots or agents. In order to perform complex tasks by means of the cooperation of several different robotic systems (i.e. different vendors and platforms), a proper integration support should be arranged. This step should be performed each time it involves heterogeneous robotic platforms and must be customized accordingly to the platforms involved. To this end, JarvSis is capable to give a specific support in the context described above, due to its integration capabilities. Indeed, as described in Chapter 2, the communication model and task organization of JarvSis allows different robots to exchange information and perform several task in a structured and optimized way.

This chapter illustrates and discusses a case study represented by a specific application supported by JarvSis. In particular, after a detailed description of the scenario – the robots involved and the environment – a number of simulation results are discussed.

5.1 Scenario

Let us consider a number of terrestrial and aerial robots (UGVs/UAVs) that are located in different geographical areas, a plant for renewable energy production, composed by multiple PV modules and wind turbines, which is located (distributed) in the same areas. Let us suppose that the plant is big enough to require several UGVs/UAVs to perform different inspection. Let us assume also

that each robot is capable to establish a wireless connection to a number of suitable servers, and that a number of base-station are located around the plant, in order to gather data collected by the robots. Base stations will send the information to the cloud or to any server located at the edge of the network [6] and send requests for inspection of specific areas.

By design, the "mission" of the different robots have been organized by decomposing each single area into a number of sections (we will refer to this process as "area decomposition") that, in turn, are composed by a number of sub-areas; each robot will be able to perform a mission in a specified sub-area. The details of the area decomposition is shown in Figure 5.1.

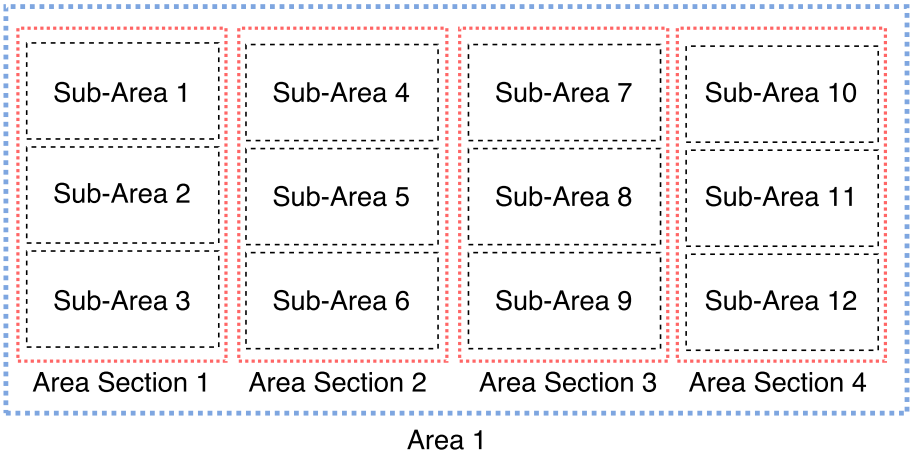

Figure 5.1: Single Area Decomposition

Then, each area section will host i) a group of heterogeneous robots (UGV/UAV) with an specific and unique ID and ii) a base station that will be able to send the requests for inspections and collect the data collected by the robots (Figure 5.2).

5.2 Simulation of the scenario

The whole scenario has been simulated by means of the AgentSimJs simulator 3. In particular, each robot has been represented, in the simulation, through the design of a specific agent. Moreover – as it happens in ROS-based software [57] – the needed communication capabilities are provided by the message buses and MQTT manager of AgentSimJs. In particular, the `MSG-BUS` manager is used to simulate communications among the agents located in the same area, while the MQTT manager is used to send/receive message with remote/external agents/platform.

The following *tasks* have been defined for each agent:

- `Go-To-Point`: invoked to drive the robot to the input destination point (x,y).
- `Scan-Area`: this task performs an area scan with the a specific sensor for a defined range (both defined as inputs).
- `Return-To-Home`: this task drives the robot to the pre-defined home position.

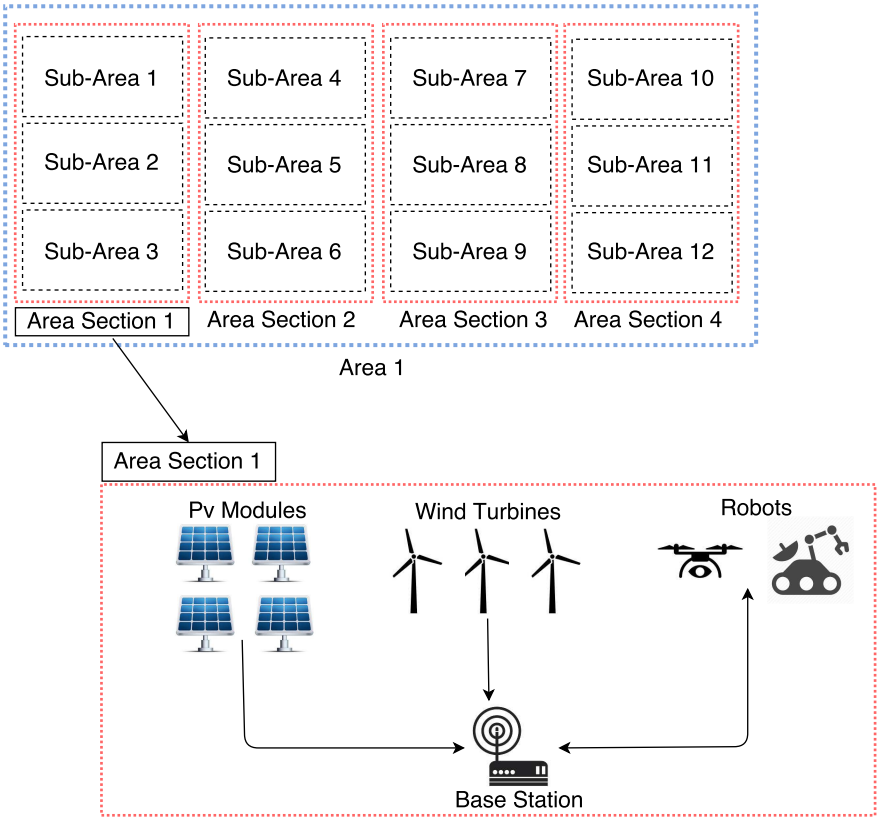

Figure 5.2: Area Section environment

- `Sync-Data`: this task is specified to push data stored in the robot to a remote storage entity.
- `Send Message`: this task is specified to send several customized message to other agents/robots within the same group.

Each robot (aerial or terrestrial) is equipped with an infrared camera capable to execute a thermographic inspection, as well as a standard camera capable to record video in parallel.

Once a base station requires an inspection of a specific point, the selected robot will reach the target area and will perform an inspection by recording a video through its own cameras. Therefore, collected data is sent to a suitable remote server in order to perform an in-depth analysis.

For instance, supposing that a given robot has to perform a mission on the sub-area no.1, it will be driven to scan a number of way-points; on this basis, the user must define a cluster composed by the needed tasks, as shown in Figure 5.3.

Such a mission is then composed by a hierarchical sequence of basic tasks defined by a JarvSis cluster of tasks. This cluster will be managed by the JarvSis instance that runs on a local server capable to communicate with all the robots by means of the MQTT communication channel.

As described in 3.4 the task management is performed in the JarvSis node, while the execution of the task is performed by AgentSimJs's agents through a specific component able to translate in a full-duplex way the message received/sent to it's dedicated JarvSis node.

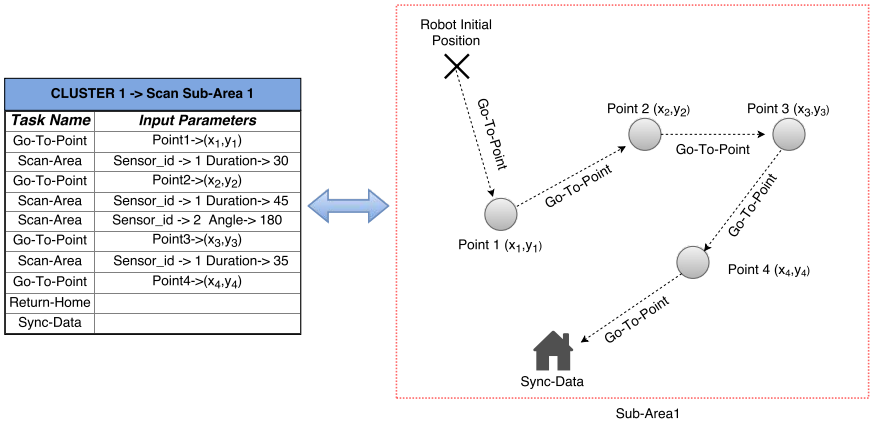

Figure 5.3: Area

5.3 Cluster and Task Organization

In order to define the clusters and related task a JarvSis network is defined for the whole plant area. This network is essentially composed by two hierarchical layers (Figure 5.4):

- A plant area layer where a single JarvSis node monitors the cluster in the Area Sections.
- An Area section layer where there is a JarvSis node for each area Section.

Moreover, the higher layer is represented by a further layer that is responsible to monitor and manage the plant area JarvSis nodes. In this way all the running processes and task can be monitored and the resilience of the system can be improved. This third layer must be able to monitor the status of all the clusters, the workload of the nodes and will perform custom management algorithms to move tasks and/or clusters from a node to another one, if needed for balancing requirements (eg. move some clusters from a plant area node to another node).

In order to simplify the simulation we simulated a single plant area composed by 4 Area Section, as illustrated in Figure 5.5. The JarvSis network topology will be composed by a single JarvSis node at plant area level and 4 “child” (JarvSis nodes) that will be placed at Area Section layer.

The inspection requests are generated by the base-station. We suppose that there is a single base station for each sub-area that generates a specific request at fixed intervals. These requests are generated through a parallel cluster defined within a JarvSis node at Area Section level.

Listing 5.1: ”Base-station Parallel Cluster Definition”

```
{
  "clusters":[
    {
      "cluster_name": "request_generator",
      "ext_application_id": "23",
      "linked_to_parent_node_task": "0",
      "ext_cluster_id": "23",
      "cluster_description": "parallel cluster to generate inspection
          request for 3 base station",
```

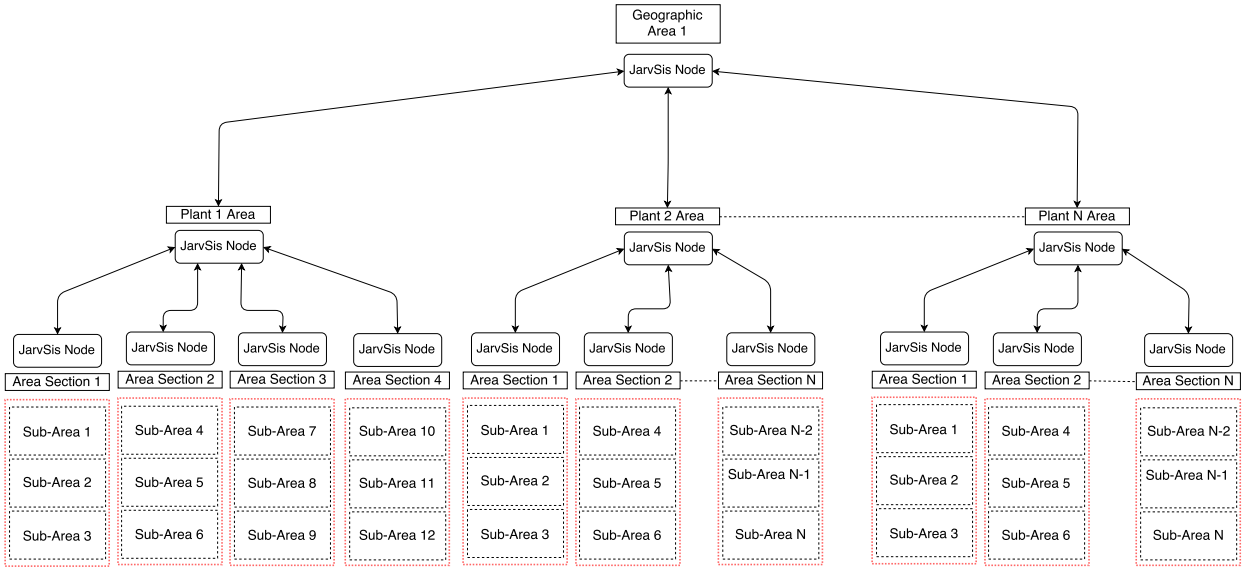

Figure 5.4: Plant Area JarvSis Network

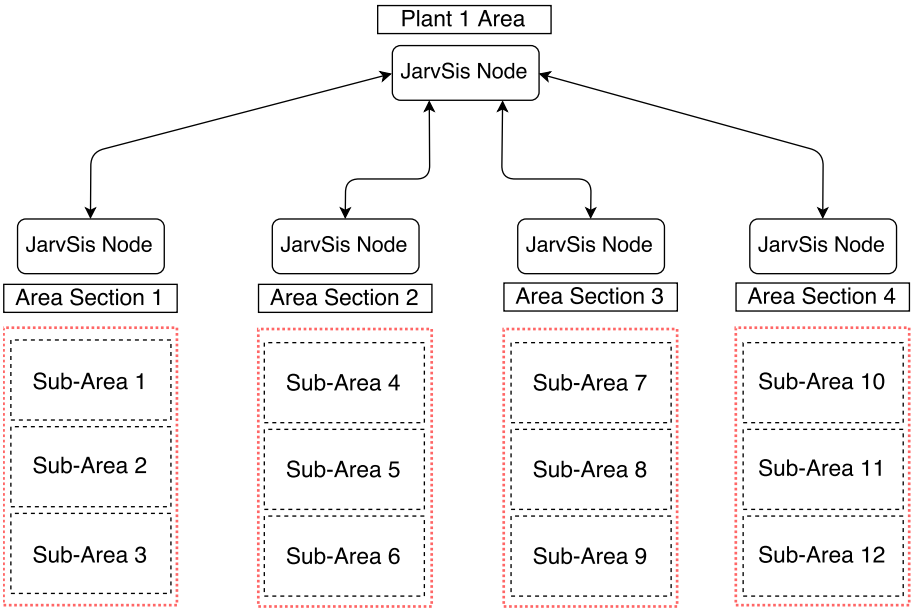

Figure 5.5: Plant Area JarvSis Network

```
9       "parent_node_task": "?",
10      "cluster_type": "parallel_cluster",
11      "ext_application_name": "agentsim_application",
12      "task_array": [
13        {
14          "task_name": "basestation_1",
15          "task_description": "generate inspection request for bs 1",
16          "ext_application_id": "23",
17          "task_priority": 0,
18          "task_starting_type": "act_timer",
19          "ext_task_id": "0",
```

```
            "task_timeout": "1000",
            "ext_application_name": "agentsim_application",
            "task_type": "external",
            "child_node_cluster": "?",
            "task_timer": "30",
            "linked_to_child_node_cluster": "0"
          },
          {
            "task_name": "basestation_2",
            "task_description": "generate inspection request for bs 2",
            "ext_application_id": "23",
            "task_priority": 0,
            "task_starting_type": "act_timer",
            "ext_task_id": "0",
            "task_timeout": "1000",
            "ext_application_name": "agentsim_application",
            "task_type": "external",
            "child_node_cluster": "?",
            "task_timer": "20",
            "linked_to_child_node_cluster": "0"
          },
          {
            "task_name": "basestation_3",
            "task_description": "generate inspection request for bs 3",
            "ext_application_id": "23",
            "task_priority": 0,
            "task_starting_type": "act_timer",
            "ext_task_id": "0",
            "task_timeout": "1000",
            "ext_application_name": "agentsim_application",
            "task_type": "external",
            "child_node_cluster": "?",
            "task_timer": "10",
            "linked_to_child_node_cluster": "0"
          }
        ]
      }
    ]
}
```

As figure 5.6 shows, JarvSis will execute the three tasks indipendently (parallel cluster). Three different tasks are defined within the "Request Scan Area" cluster and each task is directly linked to a single base station. In this way, for example, the base-station hardware and software can be minimized only to perform local data analysis on data gathered from the connected sensors. The integration with the robots and other agents will be managed by JarvSis without additional effort or customization. After a fixed time-interval, which may be different for each base station, JarvSis will send a *start-message* to the base station that, in turn, will generate an inspection request.

The messages generated by the area inspection request will be sent by JarvSis to another cluster that is responsible to select the most suitable robots and start the inspection task. The cluster *Scan Sub-Area* (Figure 5.7) is defined for each Sub-Area within a single Area Section and is composed by two sequential task: i) the *Choose Robot* task is activated by the the message generated by the *Request Scan Area* cluster and it contains a payload whit the target point selected by a specific

CLUSTER 1: Request Scan Area Sec. 1
Request Scan Sub-Area 1
Request Scan Sub-Area 2
Request Scan Sub-Area 3

Figure 5.6: Parallel Cluster request generator

base station; ii) once that the robot was selected the target point received is sent to the robot as payload within the start message generated by the *Scan Sub-Area N* task.

CLUSTER 2: Scan Sub-Area 1
Choose Robot
Scan Sub-Area 1

Figure 5.7: Sequential Cluster Scan Sub-Area

For each robot within the same Sub-Area a cluster of tasks named *Robot Mission*is defined (Figure 5.8). In our simulation, 4 different robots that can perform an inspection on a specific point, thus 4 different *Robot Mission* cluster are defined:

CLUSTER 5 -> Robot 1 Mission	
Task Name	***Input Par.***
Send Conf. Msg	
Go-To-Point	Point->(x_0,y_0)
Evaluate SoC	
Return to Home	Yes or No
Sync Data	
Update Status	SoC & Pos.

Figure 5.8: Sequential Cluster Robot Mission

In particular, the *Robot Mission* cluster is composed by the following sequential tasks:

- *Send Confirmation Message*: once the start message by the *Scan Sub-Area* cluster is received, the robot will send back to the base station a confirmation message (where the robot confirms that inspection task is accepted).
- *Go-To-Point*: the robot executes all the required action to reach the target point and perform a scan of the related area.
- *Evaluate SoC*: once the area scan is completed the robot will evaluate its *State of Charge* (the remaining energy/resource to perform other tasks) and will sent this information back to JarvSis.
- *Return-To-Home*: accordingly to the PS (state of charge) evaluated on the previous task the robot will return to home (if the PS it's below a certain threshold).

- *Sync-Data*: the gathered data during the area scanning are sent to a remote server/application for storage and further evaluation.
- *Update Status*: the robot becomes available again for new mission and sent a proper message to JarvSis.

The overall flow related to any single Sub-Area from the inspection request generation to the robot mission cluster is illustrated in Figure 5.9, while the structure of the clusters and tasks is illustrated in Figure 5.10.

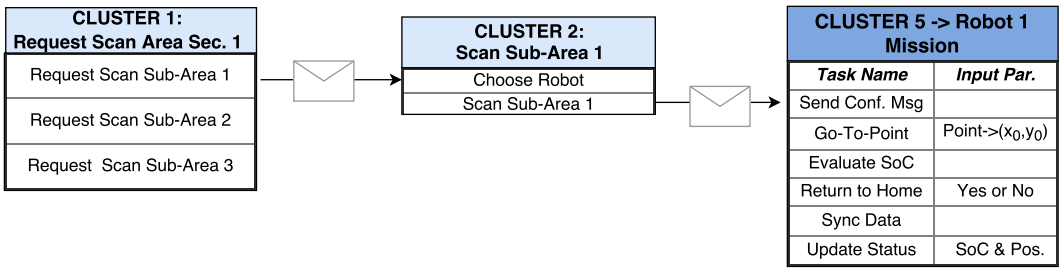

Figure 5.9: Sequential Cluster Scan Sub-Area

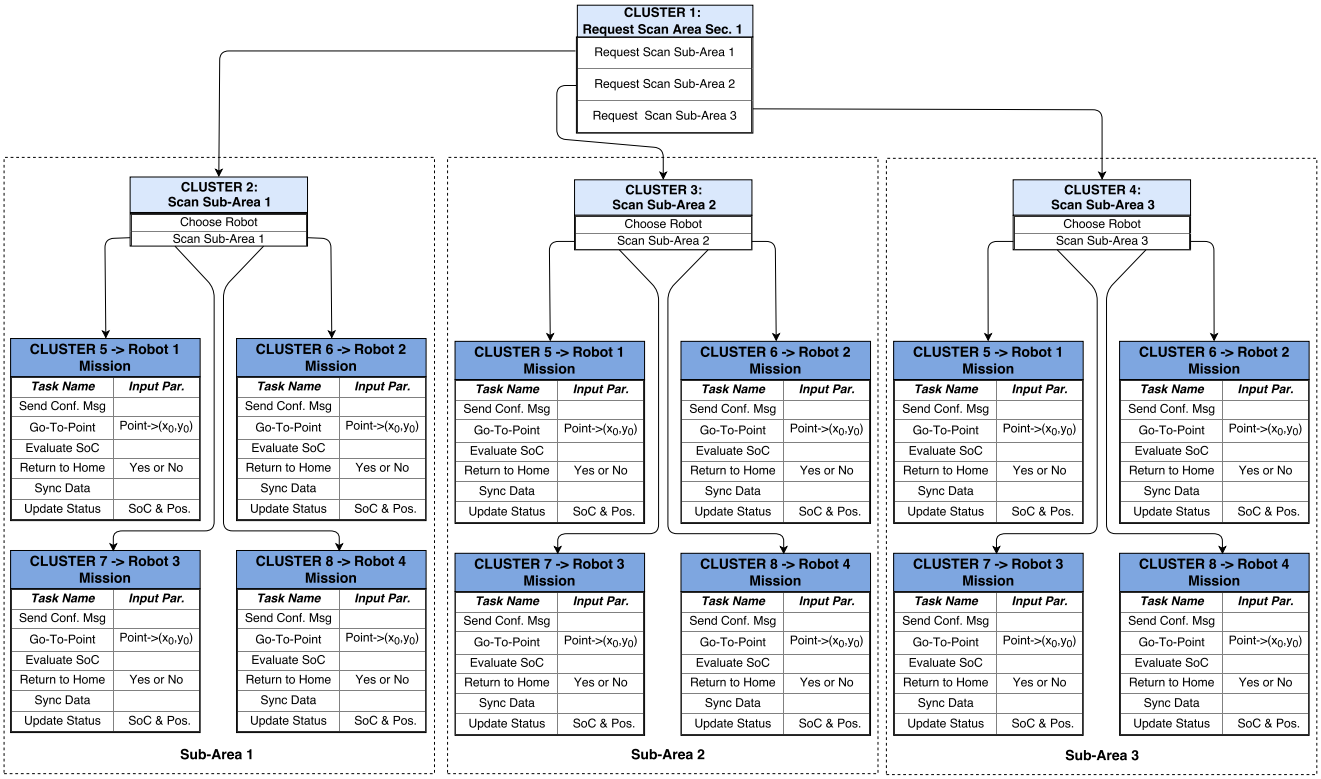

Figure 5.10: Simulation Clusters and Tasks

5.4 Robot Selection Algorithm and Environment Model

A *robot selection algorithm* has been defined for each sub-area (Figure 5.11).

The algorithm receives the target point for an inspection task by a base station, selects all the available robots (robots that are not busy in other inspection tasks) through the robot state saved in the JarvSis node Sqlite database and then proceeds with the evaluation of the task execution (if there aren't available robots the algorithm will wait until a robot will become available). To evaluate the cost of mission execution for each robot the distance between the robot and the target point is estimated and then the cost is calculated. Then the robot with the lowest cost is selected and the condition about the safety completion of the task by the selected robot is performed. If the robot is compliant with the safety completion condition then it is selected by JarvSis and the mission is assigned.

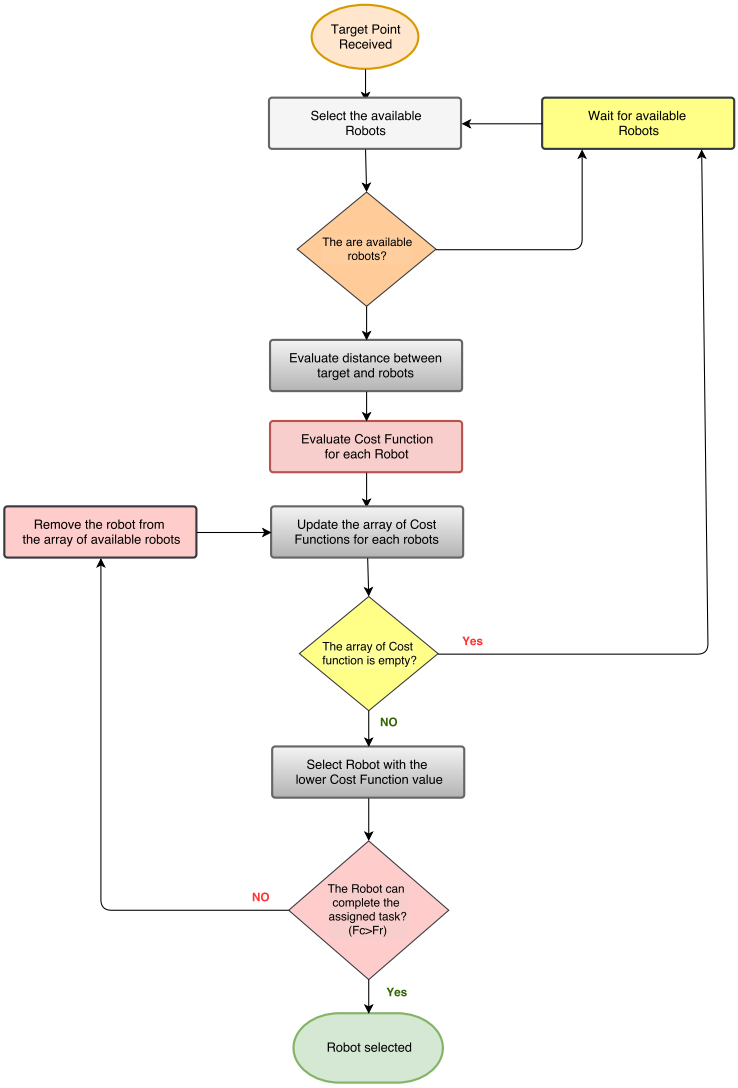

Figure 5.11: Robot Selection Algorithm

The single sub-area is composed by the following elements:

- A *Base Station* connected to the distributed sensors among the sub-area capable to generate inspection requests.
- Two UAVs capable to perform thermographic inspection through a specific camera on a target area.
- Two UGVs capable to perform thermographic inspection through a specific camera on a target area.
- Several different PV modules and Wind Turbines monitored by a number of sensors capable to communicate with the base-station.

The sub-area is mapped and represented then into a graph composed by a set of points with a pre-defined distance "d" between each others. This area decomposition is used to implement the UGV movement model, indeed the UGV cannot reach every point on the sub-area without a specific path properly planned (Figure 5.13).

A schema of a single sub-area is illustrated in Figure 5.12, where the graph of the area is represented according to the plant structure. The graph is automatically generated by a specific function implemented within AgentSimJs that removes all the points in the graph that overlap the plant components like the PV modules and the Wind turbines. The final result is saved in a dedicated array and is used as the reference graph to compute the path of UGV robots (Figure 5.12).

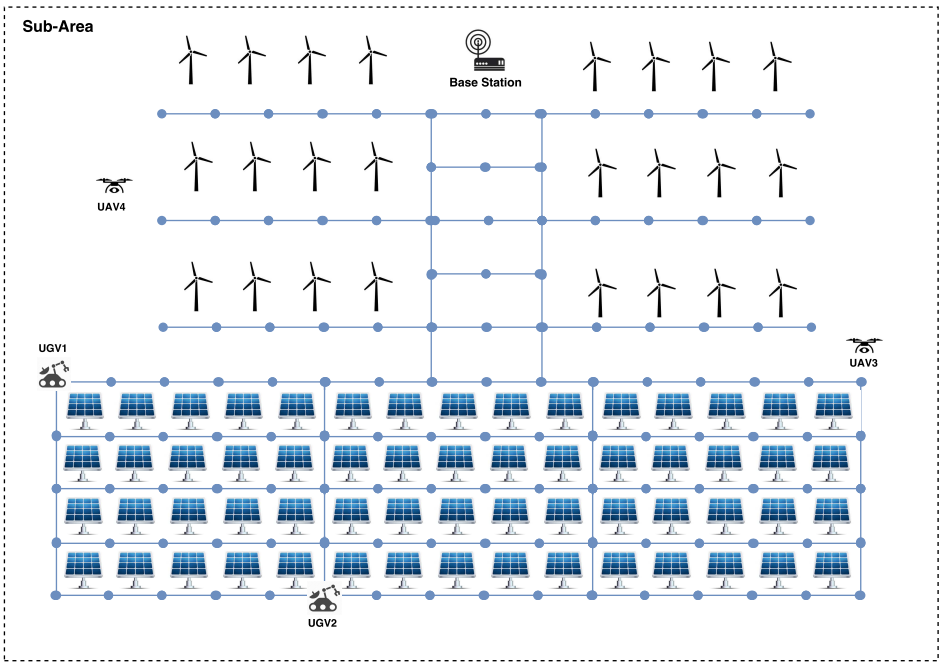

Figure 5.12: Sub Area Schema

Furthermore in order to estimate the *cost* of a task for a UGV a uniform distance estimation algorithm is used, as discussed below.

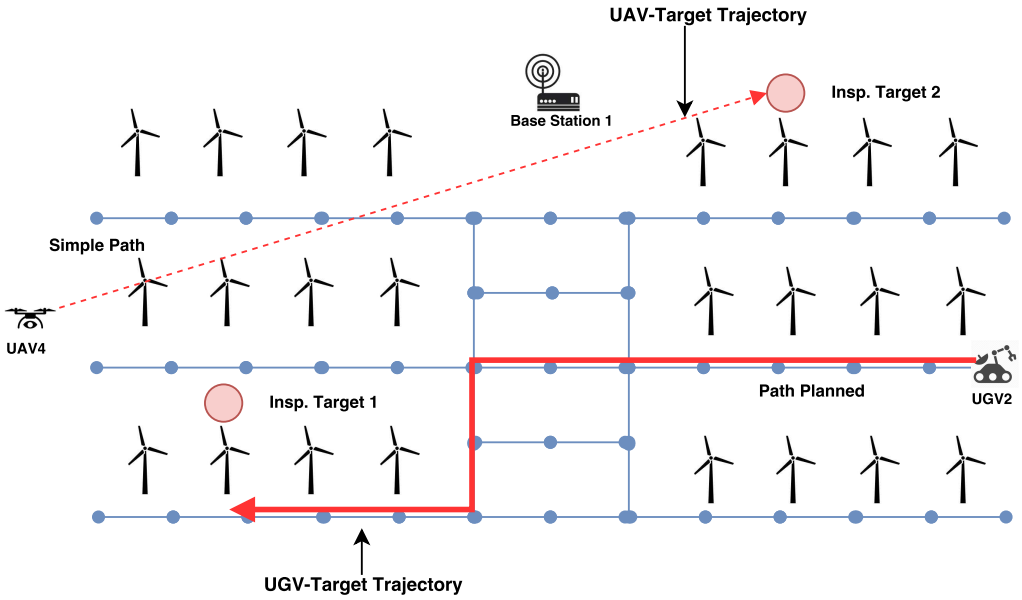

Figure 5.13: UAV and UGV trajectory

The distance evaluation is differently calculated in case the robot is an UGV or an UAV. Moreover, it is trivial for the UAV to require a specific approach for the UGV case. Assuming that the distance between each point in the graph is constant, let us denote it by d, let us denote the number of steps to reach the target point as n, then the overall distance D between two points A and B will be $n \cdot d$. This simple computation is illustrated in Figure 5.14 with n =5.

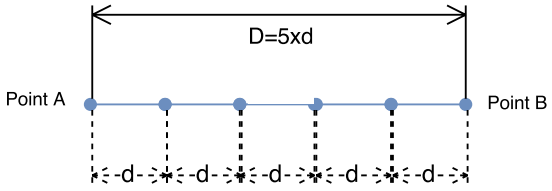

Figure 5.14: UGV distance estimation

Assuming that, for any wind turbine there is a specific point in the graph that represents the turbine itself, in the case of an inspection task for a wind turbine, the distance can be estimated by setting the Point_A as the actual robot position and Point_B as the point in the graph related to the wind turbine. Then, to estimate the number of edges n that must be "visited" to reach the target point, the Dijkstra algorithm [13] is used and then the distance D between the actual robot position and the target point is computed.

If the inspection task is related to a PV panel and there is not a correspondence between the points in the graph and the PV panels (Figure 5.15), a different approach is used, as follows ((Figure 5.16):

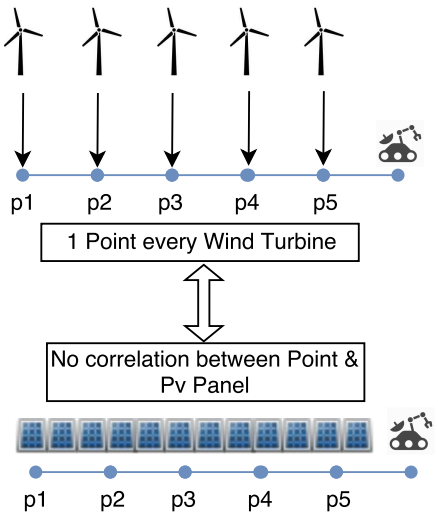

Figure 5.15: PV Panel vs Wind Turbine points correspondence

1. Find the graph point with the lower distance with the selected PV panel p1.
2. Project the PV panel target point on the graph and compute d_p.
3. Evaluate the number of edges n that must be traversed to reach the target point $p1$.
4. Evaluate the overall distance as:

$$D = n \cdot d + d_p \tag{5.1}$$

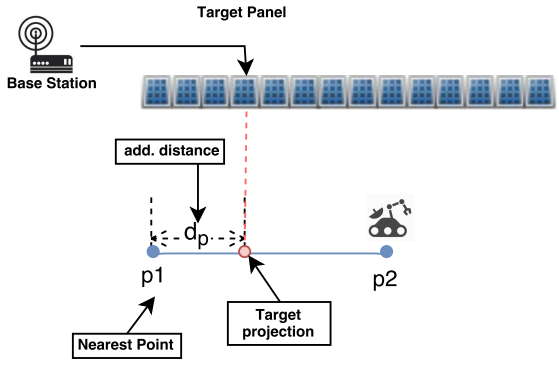

Figure 5.16: UGV target distance for PV panel target

To evaluate the cost of reaching the target point for each robot the following function is used:

$$F_c = S - c \cdot D - H_c - I_c \tag{5.2}$$

Where S represents the current battery level, c is the battery discharge rate (%c consumed every 1 distance unit), I_c is the energy consumption during the scanning phase and H_c is the

residual cost to return to home for charging the battery. H_c must be taken in consideration to assure that once the task is completed the robot can reach the home position with the residual S. This implies that a robot can complete the task only if $F_c > 0$, furthermore to enhance the resilience of the system a minimum value F_r of F_c can be introduced. F_r can be used to compensate the error related to energy consumption estimation during the movement phase or scanning phase. This means that to evaluate the capability of a robot to complete a specific task the condition $F_c > F_r$ must be satisfied.

5.5 Simulation of PV Plant inspection through UGVs and UAVs

In this section the simulation is described by focusing on the AgentSimJs tasks implementation, the scene definition and JarvSis-AgentSimJs interaction. The scene is described in AgentSimJs and it's composed by:

- A set of wind turbines in a specific area.
- Two different areas where PV modules are placed.
- Two UGV.
- Two UAV.

The PV modules and the wind turbines are defined through two specific functions. Another function is used to define the robots texture through a threejs [20] object loader that it's able to load a JSON texture created with Blender. The models of the PV and Wind Turbines are loaded before the robots textures because the robots JSON model require a callback sequence to be properly loaded (due to the complexity of the textures) (Listing 5.2):

Listing 5.2: AgentSimJs scene definition

```
function init_windturbine() {
                loader.load('model/wind_turbine.json', function(object) {
                var size = 30;
                        for (var i = 0; i < 8; i++) {
                                var clone = object.clone();
                                clone.scale.set(size, size, size);
                                clone.position.set((i*700)-2500, 0, -2500);
                                scene.add(clone);

                                //animations
                                pale.push(clone.children[2]);
                        }

                        for (var i = 0; i < 8; i++) {
                                var clone = object.clone();
                                clone.scale.set(size, size, size);
                                clone.position.set((i*700)-2500, 0, -900);
                                scene.add(clone);
```

```
19
20                              //animations
21                              pale.push(clone.children[2]);
22                      }
23
24              });
25      }
26
27      function init_PvPlants(){
28
29
30              var plant1 = new plant_obj(1,400,150,350);
31              plant1.build_plant();
32
33              var plant2 = new plant_obj(1,400,-1550,350);
34              plant2.build_plant();
35
36              plant1.build_local_monitoring(0,900,0,scene);
37      }
38
39
40      function loadobj(){
41
42      init_windturbine();
43      init_PvPlants();
44
45      loader.load('model/rover.json', function(object) {
46              robot1 = object;
47
48              var size = 50;
49              object.scale.set(size, size, size);
50              object.position.set(-400, 7.1, 90);
51
52              //animations
53              for (var i = 1; i <= 4; i++) {
54                      tmp1 = object.children[i];
55                      ruote.push(tmp1);
56              }
57
58
59      scene.add(object);
60
61      loader.load('model/drone.json', function(object) {
62                      robot2 = object;
63                      var size = 50;
64                      object.scale.set(size, size, size);
65                      object.position.set(-400, 50, 90);
66
67                      //animations
68                      for (var i = 1; i <= 4; i++) {
69                              tmp1 = object.children[i];
70                              ali.push(tmp1);
71                      }
72
73
74              scene.add(object);
75              singleAgentTest()
76              });
```

```
77          });
```

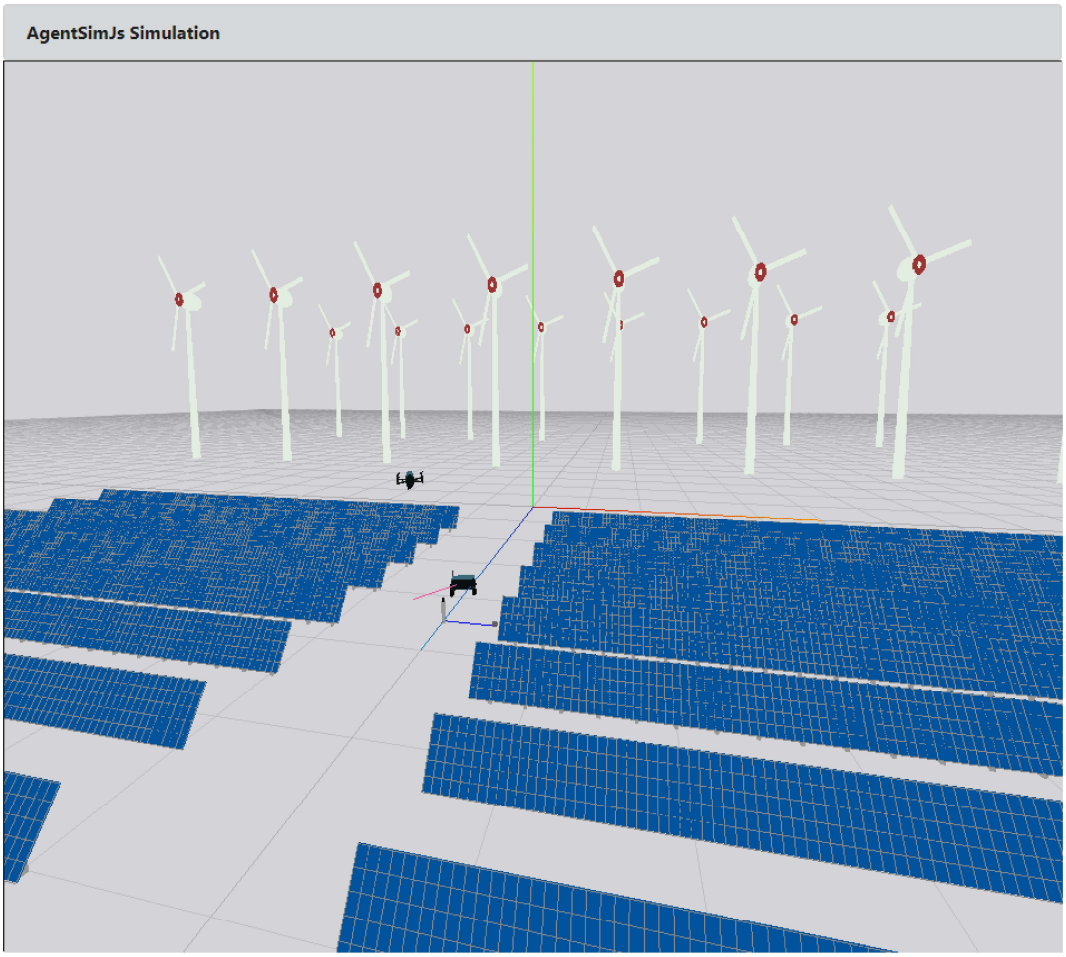

Figure 5.17: 3D scene with Threejs and AgentSimJs

As shown in Listing 5.3, the MQTT message received by AgentSimJs from JarvSis are composed by a JSON payload, to parse the received data a set of standard parameters (defined within JarvSis and shared with the AgentSimJs and all the other external application that must be involved):

Listing 5.3: Base-station request generation

```
1   //cluster json parsing
2   var cluster_ext_cluster_id="ext_cluster_id";
3   var cluster_ext_application_name="ext_application_name";
4   var cluster_ext_application_id="ext_application_id";
5   var cluster_cluster_name="cluster_name";
6   var cluster_cluster_description="cluster_description";
7   var cluster_linked_to_parent_node_task="linked_to_parent_node_task";
8   var cluster_parent_node_task="parent_node_task";
9   var cluster_cluster_type="cluster_type";
10  var cluster_task_array="task_array";
11
12  //task json parsing
13  var  task_ext_application_name="ext_application_name";
14  var  task_ext_application_id="ext_application_id";
15  var  task_task_name="task_name";
16  var  task_task_description="task_description";
17  var  task_cluster_id="cluster_id";
18  var  task_task_id="task_id";
19  var  task_ext_id="task_ext_id";
20  var  task_task_priority="task_priority";
21  var  task_task_type="task_type";
22  var  task_linked_to_child_node_cluster="linked_to_child_node_cluster";
23  var  task_child_node_cluster="child_node_cluster";
```

```
24      var  task_task_starting_type="task_starting_type";
25      var  task_task_timeout="task_timeout";
26      var  task_task_timer="task_timer";
27      var  task_payload="task_payload";
28      var  task_status="task_status";
```

In order to process the MQTT message sent by JarvSis within AgentSimJs environment, a custom processing function is defined. This function is able to process the received information and launch the requested AgentSimJs task, that implements the behaviors of every single agents, through the `ext_task_id` parameter. For example a dedicated function can be used to process the MQTT messages related to the inspection task generation and then to launch another specific function that will generate the inspection request for each base station (Listing 5.4):

Listing 5.4: Custom MQTT message processing function

```
1
2         //set custom MQTT processing function
3         mqttWsmang.setOnMessageArrived(custom_mqtt_msg_processing);
4
5         //customize mqtt msg processing
6         function custom_mqtt_msg_processing(message) {
7                 ..............
8                 ..............
9                 var msg = JSON.parse(message.payloadString);
10                //console.log("sender id:" + msg.sender_id);
11                var msg_replace="</br>" + message.payloadString.replaceAll("
                      ","," <br> ");
12
13                addMessage(msg_replace,"log2");
14
15                if((message.destinationName==sub_tpc)&&(msg.sender_id!=
                      sender_id)){
16
17
18                //console.log(msg);
19
20                        if(msg.msg_type=="START_MSG"){
21                        //console.log(msg.task_id);
22                        // each task_id is referred to a specific base
                              station
23                        if((msg.ext_task_id=="0" || msg.ext_task_id=="1" ||
                              msg.ext_task_id=="2") && msg.ext_application_id==
                              "23"){
24                                        basestationRndPos(msg.task_id,msg.
                                              ext_task_id);
25                        }
26
27
28
29
30                }
31              ..............
32              ..............
33
34
35
36        }
```

To simulate the behaviors of a single base station within AgentSimJs the following actions are defined:

1. Select the right base-station, set its status to *busy* and send a monitoring message to JarvSis to confirm that the task is running.
2. Generate the target point where the inspection must be performed and sent an activation message to the robot selection cluster.
3. Send a monitoring message to JarvSis to communicate the end of the task and set the base station status to ready.

The whole function that simulate a base-station that generates requests is shown in Listing 5.5

Listing 5.5: Base-station request generation

```
function basestationRndPos(jarvsis_task_id,local_task_id){


        var selected_base_station=0;
        var bs_status=0;
        var bs_x=0;
        var bs_y=0;
        //find the selected base station
        if(local_task_id=="0"){
                selected_base_station="0"
                bs_status=bs_local_task_id1_status;
                bs_x=bs1_x;
                bs_y=bs1_y;
        }

        if(local_task_id=="1"){
                selected_base_station="1"
                bs_status=bs_local_task_id2_status;
                bs_x=bs2_x;
                bs_y=bs2_y;
        }

        if(local_task_id=="2"){
                selected_base_station="2"
                bs_status=bs_local_task_id3_status;
                bs_x=bs3_x;
                bs_y=bs3_y;
        }

        //addMessage("base station : " + selected_base_station,"log1
            ");

        if(bs_status==task_status_ready){
        //generate first a monitoring message that set the task
            status to 1 (running)
        var task_mon_msg={sender_id:sender_id,
                                task_id:jarvsis_task_id,
                                        task_cluster_id:
                                            bs_jarvsis_cluster_id,
```

```
37                                                  task_ext_id: local_task_id
                                                        ,
38                                                  msg_type:
                                                        monitoring_msg_response
                                                        ,
39                                                  task_status:
                                                        task_status_running,
40                                                  task_payload: "?"
41              }
42
43              //change bs1 status tu running
44              if(local_task_id=="0"){
45                      bs_local_task_id1_status=task_status_running;
46              }
47
48              if(local_task_id=="1"){
49                      bs_local_task_id2_status=task_status_running;
50              }
51
52              if(local_task_id=="2"){
53                      bs_local_task_id3_status=task_status_running;
54              }
55
56
57              //pubblish task status message
58              console.log("base station  " + selected_base_station +" send
                    task msg status ");
59              addMessage("base station  " + selected_base_station + " send
                    task msg status","log1");
60              //console.log(task_mon_msg);
61              mqttWsmang.pubblish(sub_tpc, JSON.stringify(task_mon_msg), 0
                    , false);
62
63              //simulate a delay for real task execturion 20 sec
64
65              setTimeout(function(){
66
67                      //generate the task activation message for another
                            task (drone ispection execution)
68                      console.log("generating inspection task from base
                            station");
69                      addMessage("generating inspection task from base
                            station " + selected_base_station  ,"log1");
70                      //random generated position near the base station
                            position and range
71                      //r = (b-a)*rand() + a;
72                      var x_norm=(bs_x-bs_range)*Math.random()+bs_x;
73                      var z_norm=(bs_y-bs_range)*Math.random()+bs_y;
74                      //var payload={"x":Math.random()*100, "y":Math.
                            random()*100, "z":Math.random()*100};
75                      var payload={"x":x_norm, "y":10, "z":z_norm};
76                      addMessage("x_norm " + x_norm +" z_norm:" + z_norm
                            ,"log1");
77
78                      var task_act_msg={
79                                                          sender_id:
                                                              sender_id,
80                                                          task_id:13,
```

```
                                                    task_cluster_id:
                                                        46,
                                                    task_ext_id: 4,
                                                    msg_type:start_msg
                                                        ,
                                                    task_payload:
                                                        payload
                        }
                        console.log(task_act_msg);
                        mqttWsmang.pubblish(sub_tpc, JSON.stringify(
                            task_act_msg), 0, false);



                        //change bs status to completed
                        if(local_task_id=="0"){
                                bs_local_task_id1_status=
                                    task_status_completed;
                        }

                        if(local_task_id=="1"){
                                bs_local_task_id2_status=
                                    task_status_completed;
                        }

                        if(local_task_id=="2"){
                                bs_local_task_id3_status=
                                    task_status_completed;
                        }


                        //generate and pubblish task end msg
                        var task_end_msg={
                                        sender_id:sender_id,
                                        task_id:jarvsis_task_id,
                                        task_cluster_id:
                                            bs_jarvsis_cluster_id,
                                        task_ext_id: local_task_id,
                                        msg_type:monitoring_msg_response,
                                        task_status:task_status_completed,
                                        task_payload: "?"
                        }

                        //pubblish task status message
                        console.log("send task end msg ");
                        addMessage("base station " + selected_base_station +
                             " send task end msg","log1");
                        mqttWsmang.pubblish(sub_tpc, JSON.stringify(
                            task_end_msg), 0, false);


                        //change bs status to ready
                        if(local_task_id=="0"){
                                bs_local_task_id1_status=task_status_ready;
                        }

                        if(local_task_id=="1"){
                                bs_local_task_id2_status=task_status_ready;
```

```
129                         }
130
131                         if(local_task_id=="2"){
132                                 bs_local_task_id3_status=task_status_ready;
133                         }
134
135                         //simulate a delay for make the task ready again
136
137                         setTimeout(function(){
138
139                                 //generate the task activation message for
                                        another task (drone ispection execution)
140                                 console.log("set base station  " +
                                        selected_base_station + " and related
                                        task ready to perform another task");
141                                 addMessage("set base station  " +
                                        selected_base_station + " and related
                                        task ready to perform another task","log1
                                        ");
142
143
144                                 var task_ready_msg={
145                                                 sender_id:sender_id,
146                                                 task_id:jarvsis_task_id,
147                                                 task_cluster_id:
                                                        bs_jarvsis_cluster_id,
148                                                 task_ext_id: local_task_id,
149                                                 msg_type:
                                                        monitoring_msg_response,
150                                                 task_status:
                                                        task_status_ready,
151                                                 task_payload: "?"
152                                 }
153                         //console.log(task_ready_msg);
154                         mqttWsmang.pubblish(sub_tpc, JSON.stringify(
                                task_ready_msg), 0, false);
155
156
157                 }, 20000);
158
159                 }, 20000);
160
161
162                 }else{
163                         console.log("base station busy");
164                         addMessage("base station " + selected_base_station +
                                " is busy!","log1");
165                 }
166
167
168         }
```

This approach was replicated to implement all the required agents/components within AgentSimJs, where the behaviors of the robots and their communication capability are implemented and simulated (similarly to the base station implementation case). Through AgentSimJs we were able to reproduce the target environment and agents easily while focusing on JarvSis tasks and cluster

definition and implementation.

The simulation has highlighted the high capability of JarvSis to integrate heterogeneous platform and hardware within a complex environment. This use case contains all the high complex integration and organization problems related to the usage of a fleet composed by different robots that must interact with a specific environment and other external (non-robotics) agents.

The capability of modeling the agents behaviors at low level with a bottom-up approach implemented in JarvSis is a key features to tackle this kind of scenario. The level of abstraction used to group and describe each single agent actions and capability is linked to the nodes JarvSis layer and can be customized at every layer with a small effort by the user. The hierarchical cluster/tasks representation is also an easy and robust way to integrate and manage independently different applications distributed among several geographical areas. With the rise and continuous growing of IoT application and devices, the need of a geographical distributed integration and control platform is becoming critical and JarvSis can meet this need by design without further development or enhancement.

The suggested approach finally offer an high level of reliability of the whole system through the multi-node implementation strategy, that can use also multiple node at same level as a backup for the other nodes of the same level.

6

Conclusions and future work

This dissertation has discussed JarvSis, a distributed task scheduler capable to automate the execution of multiple heterogeneous tasks on IoT applications. JarvSis supports the integration of multiple software platforms and devices from different vendors, and it is equipped with a modular and adaptable software architecture. In particular, JarvSis is suitable to interact with any device that exposes remote interfaces to retrieve data and receive signals. The *lightweight* implementation of JarvSis – along with the adoption of the message bus technology MQTT – represents a suitable solution to support a wide range of application requirements, e.g. it can be deployed in a Linux based system with small computational capabilities.

A JarvSis network can be deployed by exploiting different resources from the Cloud, to the Fog, which is the layer that will support the smart devices that will operate in the "ground". One of the goals that has driven the design of JarvSis was to provide a semi-transparent support to invoke external services, to be mapped into JarvSis tasks. MQTT technology and JSON standard enabled the integration of different legacy applications into JarvSis. JarvSis has been successfully employed in an initial version – which relies only on .NET technologies (e.g. SignalR) – into a production system to manage and group a multitude of heterogeneous tasks.

In this dissertation, in order to assess the validity of JarvSis, two different use cases (Chapter 4 and 5) have been discussed in detail, in order to remark the integration capabilities introduced by the JarvSis architecture and network organization. In first use case (Chapter 4), a small JarvSis network that relies on a single layer, two complex integration problems were addresses: (i) an automated device deployment approach and (ii) the integration of the different device with a remote application and a local base station. Through JarvSis a number of sensors are deployed and integrated with an external application without complex (and time consuming) native integration of heterogeneous platform of different vendors. Leveraging on JarvSis interface to integrate sensors/robots/gateways/software optimize the integration process to a standard and well defined approach. Another important aspect described in Chapter 4 is the capability to integrate different sensors that rely on heterogeneous communication technology. Indeed, this scenario is very common when the area to be monitored holds a high number of devices. In this case, through the high integration capability given by the Java technology, different communication protocols can

implement the JarvSis interface, while maintaining a single management platform in the cloud.

Chapter 5 has described the integration and coordination of a set of heterogeneous robots which is controlled by a multi-layer JarvSis network. Indeed, the main requirement of the generic robotics application is represented by the interaction among several robots supplied by different vendors. In particular, robot cooperation is crucial to complete a complex and global mission, this will imply a strong need of coordination and control.

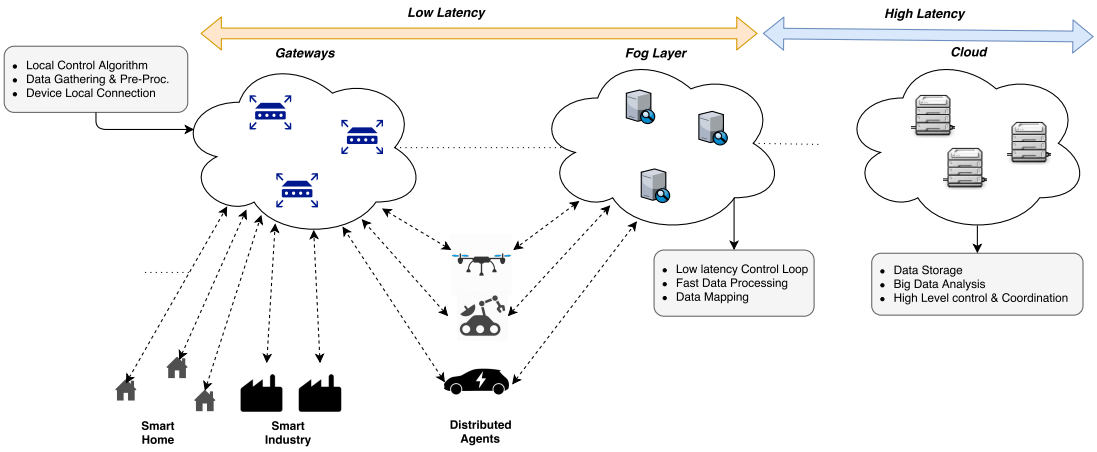

Figure 6.1: Distributed Agents Hierarchical Management

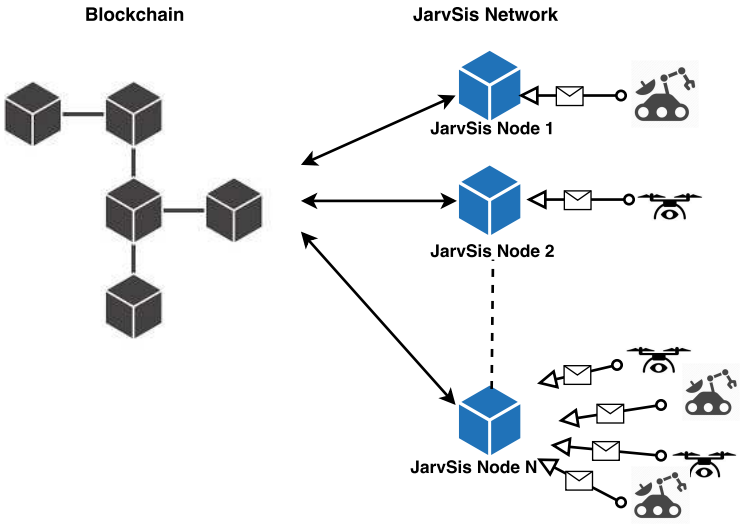

Figure 6.2: Block-chain and JarvSis network

The lack of standardization in the software used for robotic application does not facilitate the development of a single and vertically integrated solution.

JarvSis in this scenario is capable to solve this integration problem with a scalable architecture and an agnostic approach. Through the usage of a standard interface and a multi-layer JarvSis network, the mission complexity can be addressed by the design of simple and basic tasks that execute on the robots in a coordinated manner. By means of this approach, the robot vendors

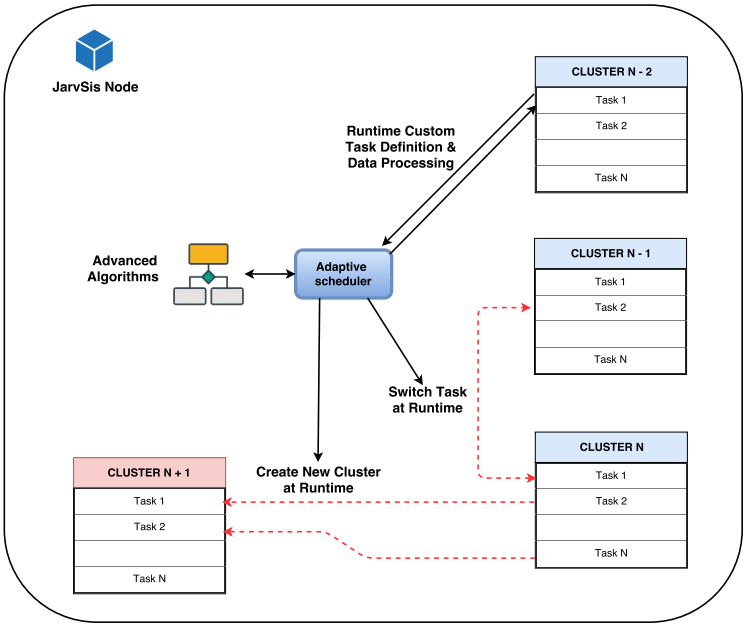

Figure 6.3: Enhanced Adaptive Scheduler

have only to integrate the JarvSis interface and implement the basic task interfaces in the robot platforms, while a monitoring platform can rely on JarvSis to monitor the robots during the mission and organize the mission according to the specific scenario.

A JarvSis network can address the problems related to the scalability of the employed IoT platforms and the heterogeneous scenarios that must be managed in geographically distributed applications. A JarvSis network can be deployed in a FOG layer as described in Chapter 5 where a layer that follows the distribution of renewable plant on field is represented. The JarvSis network described in Chapter 5 is capable to integrate all the information that come from the robots on the field, manage and optimize the robots fleet and monitor the missions status in each plant.

Typically to develop or adapt an application for a scenario where also a FOG layer must be used (multi layer application distributed among Cloud,Fog and edge layer 6.1) a specific software layer and architecture must be developed. The native multi layer structure of a JarvSis Network in this scenario can be used to develop the additional required FOG level optimizing the existing platform capability and functionality. Through JarvSis the effort needed to adapt an existing application to a multi-layer scenario and hosting system is minimized. In the near future many application developed for a classic two layer architecture (Cloud - Edge) must evolve in a three-layer architecture due to the growing concentration of device/asset for each geographical area. Through the proposed approach and technology this transition can be smoothly and fast for every software application or platform involved.

6.1 Future Work

An interesting future work would be represented by extending the implementation of the JarvSis core in order to be able to interface JarvSis with block-chain-based applications 6.2. Indeed, the machine-to-machine transactions is a new interesting area that can impact transversely several research area. Block-chain [51] platform as [28] are built to facilitate the machine-to-machine payment or transaction in a scenario where the robots/machine/software can execute economic transaction on block-chain to purchase or sell services.

The most simple example is an autonomous electric vehicle that can go to a specific recharge point while the owner is at the office or at home, in this case the car must be able to pay for the recharge service autonomously and the block-chain technology is one of the most promising candidate to handle this kind and volume of transaction.

Another important area where JarvSis can be used and can be improved is the dynamic cluster/task management 6.3. The native adaptive scheduler, used to balance the nodes workload at this stage, can be enhanced to implement more complex algorithm. Moreover, we are working to use JarvSis in the Robotic domain, in particular to manage all aspects of the missions performed autonomously by UAVs (Unmanned Aerial Vehicles) [64] and UGVs through the usage of machine learning algorithms.

Bibliography

[1] Gianluca Aloi, Giuseppe Caliciuri, Giancarlo Fortino, Raffaele Gravina, P. Pace, Wilma Russo, and Claudio Savaglio. Enabling iot interoperability through opportunistic smartphone-based mobile gateways. *J. Network and Computer Applications*, 81:74–84, 2017.

[2] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[3] Maurice J Bach et al. *The design of the UNIX operating system*, volume 1. Prentice-Hall Englewood Cliffs, NJ, 1986.

[4] A. Banks and R. Gupta. Mqtt version 3.1.1, 2014. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`.

[5] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308, 2006.

[6] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[7] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems*, 56:684–700, 2016.

[8] Susan Cheung and Vlada Matena. Java transaction api (jta). *Sun Microsystems*, 901, 2002.

[9] Simon Coakley, Marian Gheorghe, Mike Holcombe, Shawn Chin, David Worth, and Chris Greenough. Exploitation of high performance computing in the flame agent-based simulation framework. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 538–545. IEEE, 2012.

[10] Antonello Comi, Lidia Fotia, Fabrizio Messina, Giuseppe Pappalardo, Domenico Rosaci, and Giuseppe ML Sarné. An evolutionary approach for cloud learning agents in multi-cloud distributed contexts. In *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 99–104. IEEE, 2015.

[11] Antonello Comi, Lidia Fotia, Fabrizio Messina, Giuseppe Pappalardo, Domenico Rosaci, and Giuseppe ML Sarné. Using semantic negotiation for ontology enrichment in e-learning multi-agent systems. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on*, pages 474–479. IEEE, 2015.

[12] Adaptive Computing and Green Computing. Torque resource manager, 2016. `http://www.adaptivecomputing.com`.

[13] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[14] M De Benedetti, F D'Urso, G Fortino, F Messina, G Pappalardo, and C Santoro. A fault-tolerant self-organizing flocking approach for uav aerial survey. *Journal of Network and Computer Applications*, 96:14–30, 2017.

[15] M De Benedetti, F Messina, G Pappalardo, and C Santoro. Jarvsis: a distributed scheduler for iot applications. *Cluster Computing*, pages 1–16, 2017.

[16] Massimiliano De Benedetti, Fabio D'Urso, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. Self-organising uavs for wide area fault-tolerant aerial monitoring. In *WOA 2015*, pages 135–141, 2015.

[17] Massimiliano De Benedetti, Fabio D'Urso, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. Uav-based aerial monitoring: A performance evaluation of a self-organising flocking algorithm. In *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 248–255. IEEE, 2015.

[18] Massimiliano De Benedetti, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. Web-based simulations of multi-agent systems. *SIMULATION*, 93(9):737–748, 2017.

[19] Pasquale De Meo, Fabrizio Messina, Domenico Rosaci, and Giuseppe ML Sarné. An agent-oriented, trust-aware approach to improve the qos in dynamic grid federations. *Concurrency and Computation: Practice and Experience*, 27(17):5411–5435, 2015.

[20] Jos Dirksen. *Learning Three. js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd, 2013.

[21] Eric Elliott. *Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries*. " O'Reilly Media, Inc.", 2014.

[22] Subha P Eswaran and Jyotsna Bapat. Service centric markov based spectrum sharing for internet of things (iot). In *Region 10 Symposium (TENSYMP), 2015 IEEE*, pages 9–12. IEEE, 2015.

[23] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[24] Giancarlo Fortino, Alfredo Garro, and Wilma Russo. Achieving mobile agent systems interoperability through software layering. *Information and software technology*, 50(4):322–341, 2008.

[25] Giancarlo Fortino, Antonio Guerrieri, Wilma Russo, and Claudio Savaglio. Integration of agent-based and cloud computing for the smart objects-oriented iot. In *Computer Supported Cooperative Work in Design (CSCWD), Proceedings of the 2014 IEEE 18th International Conference on*, pages 493–498. IEEE, 2014.

[26] Giancarlo Fortino and Wilma Russo. Using p2p, grid and agent technologies for the development of content distribution networks. *Future Gener. Comput. Syst.*, 24(3):180–190, March 2008.

[27] Giancarlo Fortino, Wilma Russo, and Corrado Santoro. Translating statecharts-based into bdi agents: The dsc/profeta case. In *German Conference on Multiagent System Technologies*, pages 264–277. Springer, 2013.

[28] IOTA Foundation. Iota white paper - the tangle. `https://iota.org/IOTA_Whitepaper.pdf`, 2017.

[29] Python Software Foundation. The python language reference manual, 2016. https://docs.python.org/3/reference/index.html.

[30] RASPBERRY PI FOUNDATION. Raspberry pi zero. https://www.raspberrypi.org/products/raspberry-pi-zero/.

[31] The Mozilla Foundation. Indexeddb api, 2016. `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API`.

[32] Tim French, Nik Bessis, Fatos Xhafa, and Carsten Maple. Towards a corporate governance trust agent scoring model for collaborative virtual organisations. *International Journal of Grid and Utility Computing*, 2(2):98–108, 2011.

[33] IndependIT Integrative Technologies GmbH. Bic suite, 2016. http://independit.com/bicsuite/.

[34] IndependIT Integrative Technologies GmbH. Schedulix – "open source enterprise job scheduling", 2016. http://www.schedulix.org.

[35] Shweta Gupte, Paul Infant Teenu Mohandas, and James M Conrad. A survey of quadrotor unmanned aerial vehicles. In *Southeastcon, 2012 Proceedings of IEEE*, pages 1–6. IEEE, 2012.

[36] Masayuki Higashino, Tadafumi Hayakawa, Kenichi Takahashi, Takao Kawamura, and Kazunori Sugahara. Management of streaming multimedia content using mobile agent technology on pure p2p-based distributed e-learning system. *International Journal of Grid and Utility Computing 26*, 5(3):198–204, 2014.

[37] IBM Inc. Introduction to platform lsf. http://www.ibm.com/support/knowledgecenter/SSETD4_9.1.3/lsf_foundations/lsf_introduction_to.html.

[38] ECMA International. Introducing json. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[39] Jon Klein. Breve: a 3d environment for the simulation of decentralized systems and artificial life. In *Proceedings of the eighth international conference on Artificial life*, pages 329–334, 2003.

[40] Franziska Klügl and Ana LC Bazzan. Agent-based modeling and simulation. *AI Magazine*, 33(3):29, 2012.

[41] Charles M Macal and Michael J North. Agent-based modeling and simulation. In *Winter simulation conference*, pages 86–98. Winter simulation conference, 2009.

[42] Fabrizio Messina, Giuseppe Pappalardo, D Rosaci, C Santoro, and Giuseppe ML Sarné. A trust model for competitive cloud federations. *Complex, Intelligent, and Software Intensive Systems (CISIS)*, pages 469–474, 2014.

[43] Fabrizio Messina, Giuseppe Pappalardo, Domenico Rosaci, Corrado Santoro, and Giuseppe ML Sarné. A distributed agent-based approach for supporting group formation in p2p e-learning. In *Congress of the Italian Association for Artificial Intelligence*, pages 312–323. Springer International Publishing, 2013.

[44] Fabrizio Messina, Giuseppe Pappalardo, Domenico Rosaci, Corrado Santoro, and Giuseppe ML Sarné. Hyson: A distributed agent-based protocol for group formation in online social networks. In *German Conference on Multiagent System Technologies*, pages 320–333. Springer Berlin Heidelberg, 2013.

[45] Fabrizio Messina, Giuseppe Pappalardo, Domenico Rosaci, and Giuseppe ML Sarné. An agent based architecture for vm software tracking in cloud federations. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on*, pages 463–468. IEEE, 2014.

[46] Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. Complexsim: a flexible simulation platform for complex systems. *International Journal of Simulation and Process Modelling 6*, 8(4):202–211, 2013.

[47] Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. Integrating cloud services in behaviour programming for autonomous robots. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 295–302. Springer International Publishing, 2013.

[48] Fabrizio Messina, Giuseppe Pappalardo, Corrado Santoro, Domenico Rosaci, and Giuseppe ML Sarné. An agent based negotiation protocol for cloud service level agreements. In *2014 IEEE 23rd International WETICE Conference*, pages 161–166. IEEE, 2014.

[49] Michael S Mikowski and Josh C Powell. Single page web applications. *B and W*, 2013.

[50] Mahesh Nagarajan and Greys Sošić. Game-theoretic analysis of cooperation among supply chain agents: Review and extensions. *European Journal of Operational Research*, 187(3):719–745, 2008.

[51] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[52] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[53] Pasquale Pace, Gianluca Aloi, Giuseppe Caliciuri, and Giancarlo Fortino. A mission-oriented coordination framework for teams of mobile aerial and terrestrial smart objects. *Mobile Networks and Applications*, 21(4):708–725, 2016.

[54] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, et al. Argos: a modular, multi-engine simulator for heterogeneous swarm robotics. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5027–5034. IEEE, 2011.

[55] David S Platt. *Introducing Microsoft. Net*. Microsoft press, 2002.

[56] OpenLava Project. Openlava. open source workload management. `www.openlava.org`.

[57] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[58] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scheduler technologies in support of high performance data analysis. *arXiv preprint arXiv:1607.06544*, 2016.

[59] Patrick Schwab and Helmut Hlavacs. Palais: A 3d simulation environment for artificial intelligence in games. In *Proceedings of the AISB Convention*, 2015.

[60] Carfey Software. `http://obsidianscheduler.com/`.

[61] OpenPBS Team. A batching queuing system.

[62] Inc. Terracotta. `http://quartz-scheduler.org`.

[63] Seth Tisue and Uri Wilensky. Netlogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of agent*, volume 2004, pages 7–9, 2004.

[64] Kimon P Valavanis. *Advances in unmanned aerial vehicles: state of the art and the road to autonomy*, volume 33. Springer Science & Business Media, 2008.

[65] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[66] Richard Vaughan. Massively multi-robot simulation in stage. *Swarm intelligence*, 2(2):189–208, 2008.

[67] Xia Wei, Wen-Xiang Li, Cong Ran, Chun-Chun Pi, Ya-Jie Ma, and Yu-Xia Sheng. Architecture and scheduling method of cloud video surveillance system based on iot. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 551–560. Springer, 2015.

[68] Feng Xia, Laurence T Yang, Lizhe Wang, and Alexey Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101, 2012.

[69] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.

[70] Bo Zhao, Wenjie Hu, Qiang Zheng, and Guohong Cao. Energy-aware web browsing on smartphones. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):761–774, 2015.

[71] Alliance Zigbee. Zigbee specification. *ZigBee document 053474r13*, 2006.